MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

# REPORT DOCUMENTATION PAGE

| 1a. REPORT SECURITY CLASSIFICATION | 1b. RESTRICTIVE MARKINGS |
|---|---|
| unclassified | none |

**DTIC ELECTED JUL 1 1987**

**AD-A181 390**

| 3. DISTRIBUTION / AVAILABILITY OF REPORT |
|---|
| Approved for public release; Distribution Unlimited |

| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
|---|---|
| F4962086C0103041487    'D | AFOSR-TR- 87-0789 |

| 6a. NAME OF PERFORMING ORGANIZATION | 6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION |
|---|---|---|
| YLYK, Ltd. | | |

| 6c. ADDRESS (City, State, and ZIP Code) | 7b. ADDRESS (City, State, and ZIP Code) |
|---|---|
| 2440 Stone Ann Arbor, MI 48105 | AFOSR/NM Bldg 410 Bolling AFB |

| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION | 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
|---|---|---|
| USAF,AFSC,AFOSR | NM | F-49620-86-C-0103 |

| 8c. ADDRESS (City, State, and ZIP Code) | 10. SOURCE OF FUNDING NUMBERS | | | |
|---|---|---|---|---|
| AFOSR/NM Bldg 410 Bolling AFB DC  20332-6448 | PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO. |
| | 61102F | 2304 | A9 | |

**11. TITLE (Include Security Classification)**
Final Report On "Reduction of Flow Diagrams to Unfolded Form Modulo Snarls" (unclassified)

**12. PERSONAL AUTHOR(S)**
Blakley, George Robert (Bob) III

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| final | FROM 15Sep86 TO 14Feb87 | 14 Apr. 87 | 50 |

**16. SUPPLEMENTARY NOTATION**

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Graphs, GOTO, Structured Programming, Plane Embeddings, Flow Diagrams, Snarls, Object-Oriented Programming, Non-Von Neumann Architectures |
| | | | |
| | | | |

**19. ABSTRACT (Continue on reverse if necessary and identify by block number)**

This research effort produced five low-degree polynomial time algorithms for turning a complete but merely local description of a flow diagram or a graph into a standardized drawing of the entirety of that flow diagram (or graph). The most remarkable feature of these five algorithms is that all of them overcome the problem of scale. This means that the representations they produce have the property that the operation boxes and flow paths (or vertices and edges) are drawn with a uniform spacing which exhibits neither crowding nor excessive white space. Every output of every one of them is very readable. The most important of them produces a plane, straight-line drawing, without crossovers, of any planar graph. The drawing can be performed in a natural and uncrowded manner on a $3v$ by $2v$ piece of graph paper, where $v$ is the number of vertices in the graph. The research relates these graph-representation problems to fundamental problems in structured programming, especially the role of GOTOs and the uses of new non-imperative paradigms.

| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| ☒ UNCLASSIFIED/UNLIMITED  ☐ SAME AS RPT.  ☐ DTIC USERS | unclassified |

| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) | 22c. OFFICE SYMBOL |
|---|---|---|
| Dr. Arje Nachman | (202) 767-5028 | NM |

**DD FORM 1473, 84 MAR**    83 APR edition may be used until exhausted.    All other editions are obsolete.

FINAL REPORT ON
"REDUCTION OF FLOW DIAGRAMS TO UNFOLDED FORM
MODULO SNARLS"

AFOSR·TR- 87-0789

| Accesion For | | |
|---|---|---|
| NTIS CRA&I | ✓ | |
| DTIC TAB | ☐ | |
| Unannounced | ☐ | |
| Justification | | |
| By | | |
| Distribution / | | |
| Availability Codes | | |
| Dist | Avail and / or Special | |
| A-1 | | |

DTIC
COPY
INSPECTED
T

87 6 10 226

1.0                         Table of Contents

1

## 2.0 Introduction

### 2.1 Project Overview

The purpose of the Phase 1 research done on Contract F49620-86-C-0103 between Air Force and YLYK Ltd., and reported here, is to make flow diagrams more useful and comprehensible. The idea is to automate the process of drawing them to produce readable standardized objects in 2 or "2.5" dimensions (on paper) or in 3 dimensions. The fact that 3 dimensions can be required (so that a drawing on a single sheet of paper must contain at least one crossover) might seem to contradict the fundamental tenets of structured programming. But, as noted in YLYK Ltd.'s proposal leading to the Phase 1 work just completed, this is not the case. See pages 3-9 of Appendix E. See also [RU87], which raises an interesting objection to these tenets. In this Phase 1 SBIR contract, YLYK Ltd. has produced three kinds of algorithms:

1.) A "2-dimensional" algorithm (the 2jc algorithm described below) for taking a planar (i.e. imbeddable in the plane) graph and drawing it in "jailcell" fashion. This means that each edge is represented by a vertical (i.e. perpendicular to the X-axis) line segment, and each vertex (node) is represented by a "brick", namely a rectangle with short vertical edges and long horizontal (i.e. parallel to the X-axis) edges. An unexpected bonus, accruing to those who draw graphs (and hence to those who draw flow diagrams) with the jailcell algorithm, is that the "problem of scale" [EV79, p.171] disappears. This problem, which plagued people trying to do geometric (as opposed to Hopcroft-Tarjan and other merely topological) imbeddings, is that lots of vertices get jammed together when the graph is finally drawn. Users of previous geometric imbedding algorithms, therefore, needed a huge sheet of paper, and most of it was blank, but one or more small parts of it were packed with a fine tracery of vertices and edges.

2.) Three "3-dimensional" algorithms (the 3cp, 3msp, and 3mcp algorithms described below) for representing a general (i.e. not necessarily planar) graph in XYZ-space so that each vertex is a simple looking area in a level (i.e. parallel to the XY-plane) plane and each edge is an upright (i.e. perpendicular to the XY-plane) line segment. The output of each of these three algorithms resembles the skeleton of a skyscraper, in that the vertices are variously

3

shaped level floors ("terraces") and the edges are upright beams ("poles") joining floors. Neither of these algorithms suffers from the problem of scale, but this is not as surprising in 3 dimensions as it was in 2.

3.) A "2.5 dimensional" algorithm (the 2.5b algorithm described below) for making a planar drawing of a nonplanar graph. In this kind of drawing there is a jailcell configuration as in section 1. above, except that some vertex parallelograms lie above (cross over) parts parts of some. edge segments. So all crossovers in all such drawings of flow diagrams are of the form

```
                   │
                   │
                   │
                   │
                   ■
        ┌─────────────────────┐
        │                      │
        │     Do Something     │
        │                      │
        └─────────────────────┘
                   ■
                   │
                   │
                   │
                   │
```
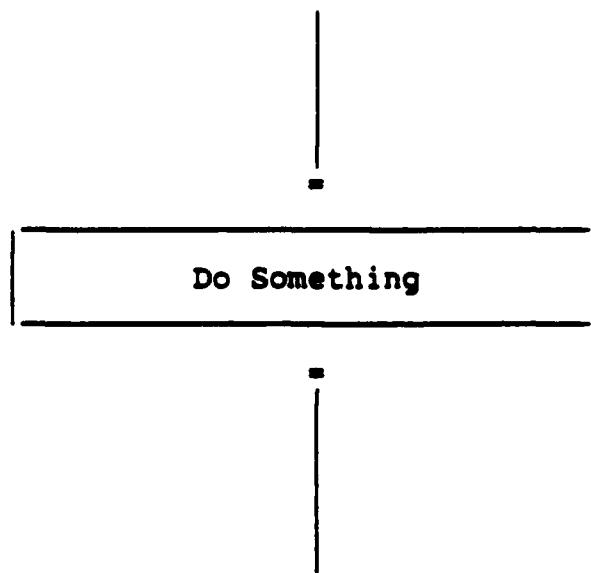
A box (or rectangle) like the above, which is wider than it is high, will be called a brick. See 1.) above. The 2.5b algorithm does not suffer from the problem of scale. At this point it is clear that no flow chart for any process whatever, and certainly no flow diagram for any proposed software, has to be drawn in the confused baroque fashion of the example on page 11 of Appendix E. From now on it will be enough to house operations in bricks and to house flows in vertical line segments. Moreover this can be done on paper (i.e. 2-dimensionally or 2.5-dimensionally) for any flow diagram, even a nonplanar one at the cost of having some vertical flows tunnel under some horizontal operation boxes.

These algorithms vary widely in depth. 2jc is a substantial advance. The others are more straightforward (though 3mcp

4

requires some analytic number theory.)

## 2.2 Objectives

Appendix A contains a detailed description of the objectives of this research effort (objectives a through h.)

## 2.3 Status of the Research Effort

As this report shows, objectives a, b, and c have been met. Objectives d and g have been partially attained, and the conditions surrounding further investigation have been well delineated for a phase 2 effort. Good preliminary progress has been made on objectives e and h; this progress can be considered a staging ground for substantial further research during phase 2. The research on objective f has not yet led to any interesting results.

## 2.4 Publications

No publications have yet arisen from this work. Several are currently in preparation.

## 2.5 Personnel

Bob Blakley, G.R. Blakley, R.D. Dixon, and A.M. Hobbs did research on this project. See pages 16, 17, 21, and 22 of appendix E for more information about these researchers.

## 2.6 Interactions

No presentations associated with this work have yet been made. No consultations have yet been entered into. No inventions or patent applications have resulted from this research.

## 3.0        Definitions of Graph-Theoretic Terms

### 3.1   Simple Graph

A simple graph G is a pair (V(G), E(G)), where V(G) is a finite nonempty set whose elements are called the vertices of G, and where E(G) is a finite, possibly empty set of unordered pairs of distinct elements of V(G). Elements of E(G) are called the edges of G.

Note that we do not consider infinite graphs in this report.

### 3.2   Graph

A graph G is a pair (V(G), E(G)), where V(G) is a finite nonempty set whose elements are called the vertices of G, and where E(G) is a finite, possibly empty bag of unordered pairs of elements of V(G). Elements of E(G) are called edges of G.

A bag is a collection of items which allows repeats; i.e. it is like a set, except that elements may appear more than once.

Note that a graph is like a simple graph, except that it may contain loops (edges from a vertex to itself) and multiple edges (more than one edge between the same two vertices -- this is why E(G) is a set in the definition of a simple graph but a bag in the definition of a graph.)

### 3.3   Connected Graph

A path from va to vb in G (where va and vb are elements of V(G)) is a finite sequence v(1),v(2),...v(n) of vertices from V(G) satisfying the following conditions:

(i)    n is a positive integer
(ii)   v(1) = va and v(n) = vb
(iii) for every v(i),v(j) such that j = i+1, {v(i),v(j)} is an element of E(G).

A graph (or simple graph) is connected if for every pair of distinct vertices va and vb in V(G), there is a path from va to vb.

### 3.4   Biconnected Graph

A graph G is biconnected if the following conditions are

satisfied:

(i) G is connected
(ii) There is no vertex v in V(G) such that the graph (V(G)-v, E(G)-e(v)) formed by removing v and all edges incident upon v is not connected.

(if v is an element of V(G), e(v) is the set

{e | e is an element of E(G) and
      e is of the form {v,v'},
        where v' is any vertex in V(G)}

.)

## 3.5 Complete Graph

A graph G is complete if every pair v and v' of vertices in V(G) are connected by an edge {v,v'} in E(G).

## 3.6 Bipartite Graph

A graph G is bipartite if the vertices in V(G) can be partitioned into two disjoint sets V1(G) and V2(G) such that every edge in E(G) is of the form {v1,v2}, where v1 is an element of V1(G) and v2 is an element of V2(G).

## 3.7 Complete Bipartite Graph

A graph G is a complete bipartite graph if the following conditions are satisfied:

(i) G is bipartite.
(ii) for any two elements v1 and v2 of V(G) such that v1 is an element of V1(G) and v2 is an element of V2(G), there exists an edge of the form {v1,v2} in E(G).

## 3.8 Digraph

A digraph is just like a graph except that the edges are ordered pairs rather than unordered pairs.

## 3.9 Planar Graph

A planar graph is a graph which satisfies the following conditions:

(i) The vertices of G can be represented by pairwise

7

disjoint points in the plane.

(ii) The edges of G can be drawn as simple curves in the plane.

(iii) For each edge {v,v'} the curve representing (v,v') has as its endpoints the points representing the vertices v and v'.

(iv) For each pair of edges {v,w} and {v,w'} which share an endpoint v, {v,w} and {v,w'} intersect at the point representing v and at no other points.

(v) For each pair of edges {v,v'}, {w,w'} which do not share a common endpoint, the curves representing {v,v'} and {w,w'} do not intersect.

Intuitively, these conditions amount to requiring that we be able to draw the graph in the plane in such a way that the vertices do not overlap and the edges do not cross one another or cross over a vertex which they do not contain as an endpoint.

3.10    Plane Imbedding

A plane imbedding of a planar graph is a mapping PE which takes vertices in V(G) to points in the plane and edges in E(G) to simple curves in the plane such that the points and curves satisfy the conditions enumerated in the definition of a planar graph above.

# 4.0 History I: Determining Graph Planarity

## 4.1 The Problem

Given an arbitrary graph, we would like to be able to tell whether or not that graph can be drawn in the plane (for example, on a sufficiently large sheet of paper), in such a way that all vertices occupy disjoint points and no two edges intersect one another except possibly at their mutual endpoints.

From this point on, this report will deal only with biconnected graphs. For arbitrary graphs, planarity determination and planar imbedding are easily accomplished by breaking the graph down into biconnected components, solving the problem for those components, and combining the solutions.

## 4.2 Early Algorithms

### 4.2.1 Euler's Formula

In 1750, Euler proved that for any plane graph, if $F(G)$ denotes the set of faces of the graph, then

$$|F(G)| + |V(G)| - |E(G)| = 2$$

Euler's proof can be found in any graph theory text (e.g. [WI79].)

A corollary of this formula is that any connected simple planar graph satisfies the inequality

$$|E(G)| <= 3*|V(G)| - 6$$

(Because any face in such a graph must have at least three bordering edges, and every edge borders exactly two faces.)

This inequality gives us a simple crude test for rejecting a large class of graphs which are not planar, but it does not allow us to answer the planarity question for general graphs.

### 4.2.2 Kuratowski $|V(G)|**6$ Algorithm

In 1930, Kuratowski discovered a necessary and sufficient condition for planarity of a connected simple graph. He proved that a graph is planar if and only if it contains no

9

subgraph which is homeomorphic to K(3,3) or K(5). A proof
of Kuratowski's theorem appears in [BM76].

Kuratowski's theorem can be used to show that a graph is
planar if and only if it contains no subgraph which is
contractible to K(3,3) or K(5) (see [WI79] for details),
which allows us to produce an algorithm for checking
planarity. The contraction algorithm, however, is
inefficient; according to Tarjan [TA71], it requires a
minimum of $|V(G)|^{**}6$ time.

## 4.3 Linear Time Algorithms

## 4.3.1 Lempel-Even-Cederbaum Vertex Addition Algorithm

Lempel, Even and Cederbaum demonstrated an algorithm for
testing graph planarity in 1966 [LE67]. This algorithm was
proved to be realizable in linear time by Even and Tarjan
[ET76] and Booth and Leuker [BL76] in 1976. A simple
presentation of the algorithm appears in [CN85].

Their algorithm works basically as follows: assume that we
have a graph G, and that some portion of G is already
imbedded, so that a number of vertices vl...vi are drawn,
and all edges connecting vertices in the set {vl, ..., vi}
(but no edges connecting vertices outside this set, and no
edges connecting vertices in the set with vertices not in
it) have also been drawn. We want to imbed the rest of the
graph.

First, for each vertex v' in {vl, ..., vi}, add a "virtual"
vertex for every vertex vj in V(G) - {vl, ..., vi} which is
adjacent to v'. Label this virtual vertex "vj" (note that
since several vertices in {vl, ..., vi} may be adjacent to
the same vj in v(G) - {vl, ..., vi}, there may be several
virtual vertices with the same label.) Connect v' to vj
with a virtual edge.

This step results in a graph which looks like the imbedded
version of {vl, ..., vi} with a bunch of labelled virtual
vertices hanging down from it by virtual edges.

Pick the smallest virtual vertex label vJ. Choose all
virtual vertices labelled vJ. If there is only one, no work
needs to be done, so simply replace the virtual vertex with
a real vertex. Otherwise (if there are several virtual
vertices labelled vJ), try to rearrange the virtual vertices
and edges in such a way that all virtual vertices labelled

10

vJ can be superimposed (the associated virtual edges will need to be moved so that they are still connected to both of their endpoints) without causing any edges to intersect. If this can be done, then replace all virtual vertices labelled vJ with a real vertex at the location of superimposition, and turn all the virtual edges into real edges at their new "rearranged" positions.

It is important to note that the rearrangement of virtual edges may require rearrangement of the already drawn real vertices and edges as well as rearrangement of virtual edges and vertices; this requires the algorithm to be able to backtrack and modify the work it has already done in imbedding {v1, ..., vi}.

If the algorithm ever reaches a condition in which a set of virtual vertices with the same level cannot be merged, then the graph is not planar. Otherwise, the algorithm generates a plane imbedding.

### 4.3.2 Hopcroft-Tarjan Path Addition Algorithm

In 1970, John Hopcroft and Robert Tarjan invented an $|V(G)|*log(|V(G)|)$ time algorithm for determining whether or not an arbitrary biconnected graph is planar. They later refined this algorithm to achieve linear time. This linear-time algorithm formed the basis for Tarjan's doctoral dissertation [TA71], and was published in [HT74]. This paper was corrected in [DE76]. A good presentation of the corrected algorithm appears in [RN77].

The Hopcroft-Tarjan approach is based on depth-first search. The basic approach is this:

(1) Choose a spine cycle C (a closed cycle of edges in the graph). Imbed it in the plane.

(2) Decompose the remaining edges in the graph into a set of edge-disjoint paths.

(3) Attempt to imbed each path in this set in the plane, either entirely inside or entirely outside C.

If all paths can be imbedded then the graph is planar; otherwise it is not. Obviously, some difficulties arise. The fact that a previously imbedded path (say, inside C) interferes with the imbedding of the current path does not necessarily imply that there is no planar imbedding; the

11

previously imbedded path might be able to be imbedded outside C instead, which would clear the way for the imbedding of the current path. It is therefore necessary to allow for backtracking in the algorithm, to permit edges to be moved from outside to inside or vice-versa if it becomes necessary. The algorithm uses depth-first search to generate paths in a canonical order and to allow for just this sort of backtracking.

## 5.0       History II: Plane Imbeddings of Planar Graphs

### 5.1   The Problem

Given a graph which is known to be planar, we would like to be able to produce a drawing of it in the plane with no edge-crossings. Recall from our definitions that a plane imbedding is described as follows:

Given a graph G, a plane imbedding is a mapping which associates a point P(v) in the plane with each vertex v of G and a simple curve C(e) in the plane with each edge e of G such that:

(1) if e = {v1,v2}, then the endpoints of C(e) are P(v1) and P(v2).

(2) no two of the simple curves representing edges intersect except possibly at their mutual endpoints.

A straight-line imbedding is, of course, a plane imbedding in which all of the simple curves produced by the mapping C are straight line-segments.

### 5.2   The Chiba-Nishizeki-Abe-Ozawa Linear Time Algorithm

Chiba, Nishizeki, Abe, and Ozawa [CN85] have modified the Lempel-Even-Cederbaum linear-time planarity-testing algorithm to produce a plane imbedding in linear time. Furthermore, they demonstrate an algorithm for finding all possible plane imbeddings of a given planar graph.

### 5.3   The Hopcroft-Tarjan Linear Time Algorithm

Rheingold, Nievergelt, and Deo note [RN77] that the Hopcroft-Tarjan planarity-checking procedure can be modified to produce a plane imbedding without requiring more than linear time. This is accomplished by modifying the algorithm so that it builds a dependency graph which reports which paths must be imbedded inside and which outside the spine cycle (and, since the checking algorithm is recursive, which paths must be imbedded inside and outside of each successive recursively generated cycle.) This dependency graph contains enough information to allow the construction of a list of clockwise traversals of each of the faces in the graph, which is one way of describing a plane imbedding. Mehlhorn [ME84] presents a complete version of the Hopcroft-Tarjan algorithm and the modifications necessary to

13

use it to generate plane imbeddings (which he calls "planar maps).

As Tarjan explains in [TA87], the first linear-time version of the Hopcroft-Tarjan planarity-testing algorithm [TA71] built the dependency graph as part of the planarity-testing process, but later versions of the algorithm were made more efficient through the addition of a data structure which made construction of the dependency graph unecessary.

## 5.4 Wagner and Fary on Straight-Line Imbeddings

Wagner [WA36] proved in 1936 that any planar graph can be imbedded in the plane in such a way that all edges of the graph can be drawn as straight lines. Fary [FA48] proved the same result independently in 1948.

Behzad and Chartrand [BC71] Present a proof of the Wagner-Fary theorem together with a description of how to produce a straight-line imbedding. The procedure they describe, however, produces an imbedding which suffers from the so-called "problem of scale"; that is, the area enclosed by the imbedding (excluding the face which contains infinity) may be very large compared to the shortest edge, even for graphs which contain relatively small numbers of vertices.

## 5.5 Duchet, et. al. on Straight-Line Imbeddings

In 1982, Duchet and four coworkers [DH83] proved a conjecture of de Fraysseix and Rosenstiehl [FR81] that every finite loopless planar graph G can be imbedded in such a way that the following conditions are satisfied:

(1) The vertices of G are represented by pairwise disjoint horizontal line-segments.

(2) The edges of G are represented by pariwise disjoint vertical line segments.

(3) The segment that represents an edge xy joining vertices x and y intersects the segments representing x and y, and has an empty intersection with all other horizontal segments.

Unfortunately, the result of Duchet, et. al., still suffers from the problem of scale; in fact, the construction presented in [DH83] compounds the scale problems of the

14

already possibly very large Fary representation. The authors do not address the scale problem in their paper.

## 5.6 The Ullman "<= |V(G)|**((log |V(G)|)**2)" Space Algorithm

Ullman [UL84] proves that planar graphs of order 4 or less (the order of a graph is the maximum number of edges incident upon any vertex) can be imbedded in the plane in area order |V(G)|*((log |V(G)|)**2). He also identifies several important families of planar graphs which can be imbedded in area order |V(G)|*(log |V(G)|). Recently, Sherlekar [SH8?] has used an approach based on Ullman's to demonstrate that plane imbeddings of planar graphs of arbitrary degree can be imbedded in area order |V(G)|**2, and that some planar graphs (indeed, even some of small degree) require area proportional to |V(G)|**2 for their planar imbeddings.

6.0    The Jailcell (2jc) Representation of Plane Graphs

6.1  Description

Recall from the previous section that de Fraysseix and
Rosenstiehl postulated, and Duchet et.al., proved, that any
loopless planar graph G can be imbedded in the plane in such
a way that the following conditions are satisfied:

(1)  The vertices of G are represented by pairwise disjoint
horizontal line segments.

(2)  The edges of G are represented by pairwise disjoint
vertical line segments.

(3)  The vertical segment that represents an edge (x,y)
joining vertices x and y intersects the segments
representing x and y, and has an empty intersection with all
other horizontal segments.

If we impose the following additional conditions:

(4)  The horizontal segments representing vertices of G are
drawn at consecutive positive integer coordinates starting
with 0.

(5)  The vertical segments representing edges of G are drawn
at consecutive positive integer coordinates starting with 0.

Then the resulting drawing is called a "jailcell imbedding".

One of the accomplishments of YLYK, Ltd. under this contract
has been the development of an algorithm to produce a
jailcell imbedding given a planar imbedding of a simple
biconnected planar graph.  This algorithm is described in
the next section.  (Actually, the algorithm described in the
next section produces vertical vertex-segments and
horizontal edge-segments, but turning this representation on
its side produces the version described above and is
entirely straightforward.)

6.2  The Jailcell Algorithm

The jailcell algorithm accepts as input a "facelist" (such
as the one produced by the modified Hopcroft-Tarjan planar
imbedding algorithm) representing a biconnected graph, and
produces a jailcell imbedding of that graph; unlike the
Hopcroft-Tarjan and Lempel-Even-Cederbaum algorithms, the

16

jailcell algorithm does not test a graph for planarity.

The jailcell algorithm is a face-addition algorithm, which differentiates it from the path-addition approach of Hopcroft and Tarjan and from the vertex-addition approach of Lempel, Even, and Cederbaum.

To run the jailcell algorithm, we need as input a plane imbedding in the form of a list of faces whose vertices are enumerated in clockwise order. "Clockwise order" is defined in the following way: pick some vertex adjacent to the face you are enumerating. Now, pretend that you are standing inside the face. Place your left hand on the "wall" (composed of the vertices and edges of the enclosing face) at the position of the vertex. Mark the vertex with chalk. Put the number of the vertex in a (previously empty) list L. Now start to walk, keeping your left hand on the wall. Every time you encounter a vertex, check to see if it is marked with chalk. If it is, stop (the list L now contains a clockwise enumeration of the vertices in the face). Otherwise, put the vertex's number at the end of the list L.

Given a plane imbedding in the form of a list of faces whose vertices are enumerated in clockwise order, we proceed as follows:

First, we choose a face to be the "exterior face" of the jailcell imbedding (i.e. the face which contains infinity.) We draw this face (representing vertices as vertical segments and edges as horizontal segments) in such a way that its interior will contain the "infinity point" of the sheet of paper we're drawing the graph on. (We can always do this because we have the vertices of the face enumerated in clockwise order.) We draw the face in such a way that no two edges adjacent to it are at the same y-coordinate and no two vertices adjacent to it are at the same x-coordinate. (We also make sure that it fits into a box whose borders are the lines y=0, x=0, y=1/2, x= 1/2.) At this point, all further drawing is constrained to take place "outside" the exterior face, which is to say "inside" the connected figure drawn on our piece of paper. It is also important to note that we draw this face in such a way that the vertical segments (representing vertices) with the largest and smallest x-coordinates have an unobstructed "visibility interval" in the interior of the figure on the piece of paper. A visibility interval between two vertical segments v1 and v2 at x-coordinates x1 and x2 (x1 < x2) is simply a vertical interval such that no vertical segment with an

17

x-coordinate x3 (x1 < x3 < x2) intrudes into the interval (so you could draw a line segment between v1 and v2 at any height contained in the interval without crossing any segment representing another vertex.) As we draw each edge of this face, we determine the unique face which is adjacent to the current face across the current edge. We place this adjacent face onto the end of an "imbedding order" list (unless it already appears in the imbedding-order list; we don't allow duplicates.)

Now we will imbed each of the remaining faces one at a time. We imbed the faces in the order in which they appear on the imbedding-order list. Get the first face off the imbedding-order list and call it F. At any point in the imbedding process, the plane (our piece of paper) is divided up into a number of faces (which have already been drawn) and a number of other "regions". Regions are areas in which faces may be drawn in the future. A region may later turn out to be a face (without having to be further modified), or it may be divided up into several different faces by the imbedding process. (Note that once a face is drawn, nothing will ever have to be drawn inside it.) We now determine which region must contain the face F. This is easy to do, since we have the clockwise ordering of the edges of F, and we can determine by examination the clockwise ordering of the edges enclosing each region in our drawing. Since each edge occurs only twice in the graph (once in each orientation), we must simply find a region R which contains any edge of F in the proper orientation, and we know that that R must contain F. (The version of the algorithm which will actually be programmed into a computer keeps a "region list" containing each remaining region and the clockwise list of its edges, so that we don't have to examine the whole graph every time we need to know which region encloses a given face.)

Now we "trace around" the face in clockwise order, starting with the edge which F has in common with R (note that F may have many edges in common with R; we simply take the first one we encounter.) as long as the edges of F we "trace over" are also edges of R, we do nothing (they are already drawn.) When we come to an edge el which is in F but not in R, then we need to do some more drawing. First, we find the longest clockwise path P anchored at el which contains no edge in R (that is, the longest sequence of edges of F, enumerated in clockwise order, starting with el, such that no edge in the sequence is in R.) Say this path is {el,...,ek}. The edge ek has the property that its

18

"counterclockwise" endpoint is shared with an edge in the path (that is, in F but not in R), but its clockwise endpoint is shared with an edge which is in both F and R. Our task now is to draw P.

To draw P, we need to find the visibility interval of the region R. We are guaranteed when we imbed the first face after the exterior face that all the regions in the plane will have visibility intervals (since there's only one region and we constructed it so that it would have such an interval.) As will be seen below, all the regions we generate during the imbedding process will have visibility intervals, so we will always be able to find such an interval for R. (A program implementing the algorithm should probably determine a region's visibility interval and store it in the region list at the time the region is created.) Once we have found the visibility interval of R, we draw P as follows:

Say that the endpoints of e1 are v1cw and v1ccw (v1ccw is the first vertex of the e1 encountered when traversing F in clockwise order and v1cw is the second vertex encountered when traversing F in clockwise order); similarly, the endpoints of e2 are v2cw and v2ccw, and so on (note that the only two vertices in the path which have already been drawn are v1ccw and vkcw. Note also that v1cw = v2ccw, v2cw = v3ccw, and so on). Assume without loss of generality that v1ccw is drawn at x-coordinate x1 and vkcw is drawn at x-coordinate xk, where x1 > xk (so v1cw is to the right of vkcw on our sheet of paper). "Extend" v1 in the y-direction (the extension will consist of a segment added to either the top or the bottom of the segment currently representing v1) to the height which is at the halfway point of the visibility interval of R. Now say that the (vertical) visibility interval of R is (yb, yt) where yb < yt. The halfway point of the visibility interval is ((yt - yb) / 2) + yb. Call this value yc. Let xc = x1. Perform the following procedure:

```
for i := 1 to k do

   draw a segment representing ei at at y-coordinate yc,
      extending from x = (3/4(xc-xk)+xk) to x = xc.
   set xc := (3/4(xc-xk)+xk).

   if (i not equal to k) then do
     draw segment representing vicw (also v(i+1)ccw)
         at x-coordinate xc
         extending from y = yc to y = ((yt-yc)/2)+yc.
    set yc := ((yt - yc)/2)+yc.
   od (* end "if" *)

od (* end "for" *)

extend vkcw by adding a segment at x-coordinate xk
   extending from y = yk to y = yc.
```

Notice that, after the path P is added using this algorithm,
R has been divided into two new regions R1 and R2, a "top"
region and a "bottom" region. Assume without loss of
generality that R1 is the "top" region and R2 is the
"bottom" region. If the vertical extent of the visibility
interval of R was Y(R), and the "end vertices" of R which
were used to determine that visibility interval were vL(R)
and vR(R), then it is easy to determine that the "top"
region R1 has a visibility inverval with end vertices vL(R)
and vR(R) and vertical extent (1/(2**k))*Y(R), and the
"bottom" region R2 has a visibility interval with end
vertices vkcw and vlccw and vertical extent (Y(R)/2).

The reader can verify that the use of the factors 1/2 and
3/4 in the formulas for positioning vertex- and
edge-segments in the procedure above guarantee that no two
edge-segments will ever occupy the same y-coordinate and no
two vertex-segments will ever occupy the same x-coordinate.

At this point we have imbedded P, but there may still be
some edges of F which have not yet been drawn. Determine
which of the regions R1 and R2 contains F (one of the two
must). Assume without loss of generality that it is R1.
Simply continue the clockwise "tracing" of F, imbedding the
next path which is not part of R1 in the same way as we
imbedded P above. Continue this path-imbedding process
until all the edges and vertices of F have been drawn. The
face-imbedding process for F is now complete. We have drawn
all of F, and nothing more will have to be drawn inside F.
We have also created a number of "other" regions, each of

which has a visibility interval of finite length, into which future faces may be imbedded.

Now repeat the face-imbedding procedure described above until all faces have been drawn.

In the final step, we must move all edges and vertices to integer coordinates. To do this, we simply sort the vertical coordinates of the segments representing edges into increasing order and assign to each coordinate the integer corresponding to its position in the resulting ordering, starting with 1. We then replace the "original" vertical coordinates in all edge-segments and vertex-segments with the "new" integer coordinates. (since no two edges occupy the same vertical coordinate, we don't have to worry about having two edges at the same integer height, and since the way we generate the vertical coordinates insures that no edge initially occupies an integer height other than 0, we don't have to worry about our "new" integer coordinates generating name-clashes with the old coordinates during the renaming process.)

We repeat the above procedure to sort the horizontal coordinates and replace them by consecutive integers.

We have now finished building the jailcell representation of the graph, and we can print out the line segments which comprise it.

In order to turn the jailcell representation of a graph into a jailcell flowchart, we must do several things: first, we must take the jailcell imbedding of the graph and "turn it on its side" so that vertices are horizontal and edges are vertical. Second, we must expand the (now) horizontal line segments representing vertices into horizontal bricks in the correct way. Third, we must expand the horizontal axis and reposition the (now) vertical segments representing edges so that those edges representing downward flows are drawn at odd x-coordinate positions and those representing upward flows are drawn at even x-coordinate positions (it is possible that this procedure may generate "gaps". A gap is a pair of consecutive integer x-coordinate locations both of which are unoccupied by edges. Gaps occur when an up-flow and a down-flow occupy adjacent x-coordinates in the underlying jailcell graph. Gaps can simply be removed, and the obvious renumbering performed. Removal of gaps results in a jailcell flowchart in which at least one of every two

21

consective-integer x-coordinate locations is occupied by a flow.) Each of these three transformations is entirely straightforward.

## 6.3 Time and Space Complexity Considerations

### 6.3.1 O-Notation

In the analysis below, we follow the time-order notation convention of Aho, Hopcroft, and Ullman [AH74]. A function $g(n)$ is said to be $O(f(n))$ if there exists a constant c such that $g(n) <= c*f(n)$ for all but some finite (possibly empty) set of non-negative values of n. In this case, we say that $g(n)$ is "order $f(n)$".

Thus, when we speak of an imbedding "occupying $O(f(|V(G)|))$ area", we mean that the amount of area needed to draw an imbedding of a graph G with $|V(G)|$ vertices is no more than some constant times $f(|V(G)|)$ units, where units are some standard measure of area such as square centimeters.

Similarly, when we speak of an algorithm "running in $O(f(|V(G)|))$ time", we mean that the algorithm will take no more than some constant times $f(|V(G)|)$ steps when run on a graph with $|V(G)|$ vertices, where "steps" are some standard time interval such as seconds or microseconds.

In what follows, "linear time" means time linear in the number of vertices of the graph (i.e. $O(|V(G)|)$.)

The important thing to bear in mind is that order statistics do not give information about the absolute running time of an algorithm (or area of an imbedding); instead they describe the growth of the running-time of an algorithm (or area of an imbedding) as a function of the number of vertices provided as input.

### 6.3.2 Space Complexity of Jailcell Representation

It should be evident from the definition of the jailcell imbedding that such an imbedding occupies $O(|V(G)|**2)$ area. (In fact, it fits exactly in a rectangle whose sides have lengths $|V(G)|$ and $|E(G)|$, where, by Euler's formula, $|E(G)| <= 3*|V(G)|-6$.)

### 6.3.3 Worst-Case Time Complexity of Jailcell Algorithm

We observe first that by Euler's formula,

$$|E(G)| <= (3 * |V(G)|) - 6$$

so $|E(G)|$ is $O(|V(G)|)$. Similarly, $|F(G)|$ is $O(|V(G)|)$.

Before we run the Jailcell algorithm, we can construct an "adjacent face list" which tells us for each edge which two faces it borders on. If we take the simpleminded approach to building this list, we could have to visit every edge in every face in our facelist (there are $2*|E(G)|$ of these) to calculate each entry of the list. Thus, building this list could take $2*(|E(G)|**2)$ time, which is $O(|V(G)|**2)$.

Drawing the exterior face takes linear time. Determining and storing the visibility interval of the interior also takes linear time, since we could have to look at every edge adjacent to the region. Updating the imbedding-order list as we proceed with the drawing takes $O(|V(G)|**2)$ time; since we can use the information in the adjacent-face list to determine which face to put on the imbedding-order list each time we visit a face, we must simply search the adjacent-face list (which has $O(|V(G)|)$ entries) for each face adjacent to the interior region. The entire procedure of adding the exterior face thus can be accomplished in $O(|V(G)|**2)$ time.

Each additional face has no more paths than it has edges (since each path has at least one edge), so we must run the path-imbedding procedure no more than $O(|V(G)|)$ times. Imbedding a path involves: (1) finding it (you might have to search all the edges in a face if there are no paths), which takes $O(|V(G)|)$ time, (2) finding which region it is in (you might have to compare all the edges of the path with each edge in each entry on the region list, which would take $O(|V(G)|**3)$ time, (3) drawing it, which takes $O(|V(G)|)$ time, (4) finding the visibility intervals of the new regions created, which also takes linear time as observed above, (5) adding the new regions to the region list, which takes linear time since we must construct their edge lists, and (6) updating the imbedding-order list at each edge, which takes $O(|V(G)|**2)$ time since we could have to search the adjacent-face list once for each edge we add. Thus, the entire path-drawing procedure takes no more than $O(|V(G)|**3)$ time, and we must repeat it no more than $O(|V(G)|)$ times per face, so imbedding a face takes no more

23

than $O(|V(G)|**4)$ time.

Finally, there are no more than $O(|V(G)|)$ faces, so imbedding the entire graph after the first face takes no more than $O(|V(G)|**5)$ time, and since building the adjacent-face list and imbedding the first face took only $O(|V(G)|**2)$ time, the entire jailcell algorithm is polynomial and in fact can be run in $O(|V(G)|**5)$ time.

The algorithm as we have stated it is almost certainly not the most efficient algorithm possible for generating jailcell imbeddings; a phase-II objective should be to produce a more efficient version (linear-time, if possible; certainly $O(|V(G)|**3)$ or even $O(|V(G)|**2)$ should be achievable.)

On page 5 of Appendix E we have given an extremely simple example of a GOTOful program, K(3,3) ZENSORT. It was designed for logical simplicity, and it was therefore reasonable to make it a sort. The reason for this is that the only operations performed in sorts are compares and data moves. This simplicity of structure enables the reader to see the necessity for an unconditional branch without distractions in a language of very little power. But it is reasonable to ask whether there are important types of problems which actually require a GOTOful approach in coding. In other words, are there types of programs which should be written with unconditional branches? A moment's reflection will suggest error trap routines, which have come to be the despair of those who (erroneously) equate structured programming with GOTOless code. The natural way to code most error correction procedures is to go about the error trapping on a case by case basis, and within each case escape to a subroutine which corrects an error in any one of a variety of ways which depend on the presumed nature of the error. In general, these error-trap routines do not return to the point of call, so a branch to such a routine is an unconditional one-way branch.

Error control was picked above because it is an outstanding example of the need for unconditional branches in current-day programs. The reader can no doubt think of numerous other types of problems which cry out for programs with unconditional branches. Rubin [RU87] appears to think that most sizable problems are most naturally handled by code involving some unconditional branching.

YLYK Ltd. takes a less extreme position than Rubin's. We merely claim that much important code (not merely error trapping routines) should be written using GOTOs. We base this position on the observation that large graphs are likely to exhibit snarls (see the extensive discussion of snarls in the proposal which led to this contract. It can be found on pages 7-10 of Appendix E). A snarl, it will be recalled, can be regarded as a minimal nonplanar piece of a flow diagram. The presence of a snarl in a flow diagram is an indication that the diagram locally departs from the Bohm-Jacopini paradigm [BJ66] of imbeddability in a plane. Their proof that there is an algorithm "equivalent" to the algorithm suggested by the flow diagram under discussion, and that the equivalent algorithm has a planar flow diagram, is irrelevant. For whatever reason, the flow diagram under

25

discussion is the one which will be reduced to code. And this diagram has a snarl. So the natural way to write the corresponding code is with a GOTO. It may well be that that the systems analyst has explicitly rejected a planar flow diagram in favor of a nonplanar one to accomplish the same objective. In other words, despite much hortatory ink spilled since [DI68], GOTO may often be the way to go.

YLYK Ltd. has devised three different ways to turn appropriate local information (i.e. the adjacency matrix, zeta, of the graph) into a global solid (i.e. 3-dimensional) flow diagram, and one way to turn it into a "2.5 dimensional" representation (i.e. like a drawing on paper, but with crossovers).

We make use of several technical terms, such as pole, terrace, brick, and lath, in what follows. See Section 7.1 below for definitions of these terms. (We have no greek font available on this word processing system. For compatibility with the original proposal contained in Appendix E, we will simply write the English name of the Greek letter zeta.)

1.) An unintelligent canonical method called the "2-level crossbar/pole" representation (3cp). This produces a "short fat" two-story structure in 3 dimensional space. There are numerous terraces on level 1 and numerous terraces on level 2, as well as numerous upright beams (poles), each of which connects one of the former to one of the latter, but touches no terrace other than these two.

2.) A largely canonical (hence largely unintelligent) method called the "multilevel starbody/pole" representation (3msp). This produces a "long tall" structure in 3 dimensional space. It has exactly one terrace on every level from level 1 to level v, where v is the number of vertices in the original graph. There are numerous upright beams (poles), each of which goes from one terrace to another without touching any other terrace.

3.) A "multilevel convexbody pole" representation (3mcp) similar to 3msp, but with convex terraces.

4.) A somewhat intelligent noncanonical method called the "basketweave" representation (2.5b). This draws a nonplanar graph on paper. the price it pays for this unnatural act is the need to draw a few crossovers. Every such crossover is of the same kind, a lath tunneling under a brick. This graphical convention is very natural and easy to take in at a glance.

## 8.1 Terminology for YLYK, Ltd.'s Graph Imbedding Algorithms

It will be necessary to describe drawings of a graph in 2 dimensions and in 3. Though these two types of drawings are in the same spirit, it will be important to have terminologies which immediately alert the reader to the dimensionality of the context.

Thus in 2 dimensions (i.e. in the XY-plane):

the X-axis will be called "horizontal";

the Y-axis will be called "vertical";

a rectangle whose width exeeds its height will be called a "brick";

a line segment parallel to the Y-axis will be called a "lath".

And in 3 dimensions (i.e. in XYZ-space):

the X-axis will be said to run "east/west";

the Y-axis will be said to run "north/south";

the Z-axis will be said to run "up/down";

a plane parallel to the XY-plane will be called "level"

a connected set of points in a level plane will be called a "terrace";

if a terrace is a rectangle with sides parallel to the X and Y axes whose north/south extent exceeds its east/west extent, it will be called a "beam";

if a terrace is a rectangle with sides parallel to the X and Y axes whose east/west extent exceeds its north/south extent, it will be called a "plank";

if a terrace is starshaped [MU75, p.330] with respect to the point at which it intersects the Z-axis, it will be called a "starbody";

if a terrace is convex it will be called a "convexbody";

a line segment parallel to the Z-axis will be called a "pole".

We thus have the analogies

    level       : pole   : up/down  : 3 dimensions ::

    horizontal : lath  : vertical : 2 dimensions.

Also, though perhaps not exactly analogous to one another, terraces and bricks are used the same way. Each of them represents a vertex in a graph, and each of them should be regarded as a piece of cardboard on which you can write an instruction in a flow diagram. Poles and laths are exactly analogous to each other and are also used the same way. Each of them represents an edge in a graph, and each of them should be regarded as a single flow in a flow diagram.

With this dimension-specific terminology we can easily describe the outputs of all our algorithms.

Let G be a biconnected graph. The output of the jailcell algorithm is the union of a set of bricks and a set of poles. The height of each brick is 1/2 and the y coordinate of the center of each brick is a positive integer no larger than $|V(G)|$. The leftmost points of any brick have x coordinate of the form $t - 1/4$, where t is a positive integer no larger than $|E(G)|$, the number of edges of the graph being represented.

The rightmost points of any brick have x coordinate of the form $s + 1/4$, where s is a positive integer no larger than $|E(G)|$.

No two bricks contain points with the same y coordinate. Hence, for every positive integer j between 1 and $|V(G)|$ inclusive, there is exactly one brick whose center has y coordinate equal to j. There are no other bricks. In crude intuitive language the bricks have uniform vertical spacing. Since the vertical extent of each brick is 1/2 it follows that the highest point of any brick is 1/2 unit lower than the lowest point of the "next brick up". These two bricks might not overlap horizontally, of course. By this we mean that that the largest x coordinate of any point of one brick might be much smaller than the smallest x coordinate of any point of the other.

29

The set of poles in the output of the jailcell algorithm has $|E(G)|$ members. Since only planar graphs G can be drawn by the jailcell algorithm, it follows [WI79] that $|E(G)|$ is no larger than $3|V(G)| - 6$. The x coordinate of any point of any pole is a positive integer no larger than $2*|E(G)|$, which is, in turn, no larger than $6*|V(G)| - 12$. Every pole intersects exactly two bricks. The intersection of any pole with any brick is a single point. It follows that every such intersection occurs at the top or at the bottom of the brick in question, as well as at the top or at the bottom of the pole in question. If the graph being represented is a directed graph then the points of a pole have odd x coordinate if the pole represents a flow from a lower brick to a higher brick, and have even coordinate if the pole represents a flow from a higher brick to a lower one.

We turn now to the output of the 2-level crossbar/pole algorithm (3cp), an algorithm so named because its outputs look something like the old crossbar switches in telephone exchanges. Let G be any graph. Then the output, in 3 dimensions, of 3cp looks like the floor planks, the roof beams, and some of the uprights joining these two types of structural members, visible a few days into the construction of a supermarket. In our agreed-upon terminology there will be $|V(G)|$ beams such that the z coordinate of every point on every beam is equal to 1. There will also be $|E(G)|$ planks such that the z coordinate of every point on every plank is 0. There will be $2*|E(G)|$ poles of up/down extent 1. If $(x,y,0)$ is the point at the base of any pole, then x is a positive integer no larger than $|V(G)|$ and y is a positive integer no larger than $|E(G)|$. In fact, the ith plank is the set

$$P(i) = \{(x,y,0) : \quad 1/2 \;<=\; x \;<=\; |V(G)| + 1/2,$$
$$i - 1/4 \;<=\; y \;<=\; i + 1/4 \quad\quad\}$$

and the jth beam is the set

$$B(j) = \{(x,y,1) : j - 1/4 \;<=\; x \;<=\; j + 1/4,$$
$$1/2 \;<=\; y \;<=\; |E(G)| + 1/2 \}$$

There are poles from $B(j)$ to $P(i)$ and $P(k)$ if and only if there is an edge from vertex i to vertex j in the graph G. If the graph G is a directed graph then each pole has an orientation imposed on it in the obvious way.

## 8.2 The 2-Level Crossbar Pole (3cp) Representation of Nonplanar Graphs

The 2-level crossbar/pole algorithm (3cp) does not faithfully draw a graph G by means of planks and beams to represent vertices, and poles to represent edges. Hence it is different from all other algorithms exhibited in this report. It goes right back to the presumed flow diagram D which was the motivation for producing zeta, the adjacency matrix of the graph G. The idea behind the research being reported here was to draw flow diagrams, not to draw graphs. In the rest of this report the way we draw a flow diagram is to draw its underlying graph. But not here. Here we start with a flow diagram D, produce the corresponding G, then use G to produce a larger graph H which also faithfully represents what D does. In fact H is the graph underlying a flow diagram E which is merely an expansion of D by the addition of some NOP (no operation) boxes.

The 3cp algorithm works as follows.

Algorithm 3CP:

[1] Start with some description which completely specifies the flow diagram D. This description will be complete, but local.

[2] On the basis of this purely local information produce the adjacency matrix zeta of the graph G which describes D.

[3] Color every vertex of G red.

[4] Use the colored version of the graph G to produce a colored graph H in the following fashion. At the middle of every edge of G put a new vertex, colored green. The graph so obtained is called H. The graph G had $|E(G)|$ edges and $|V(G)|$ vertices. The graph H has $2*|E(G)|$ edges and $|V(G)|+|E(G)|$ vertices. The graph H is not only 2-colored, it is bipartite. No green vertex is adjacent to any other. No red vertex is adjacent to any other.

Comment. (4) is a graph- theoretic way of putting a (green) NOP operation into every control flow between (red) operations in D so as to produce the equivalent flow diagram E.

[5]    There are  |V(G)|  red  vertices of the new graph  H.
Order  them in any fashion so that you have vertices  r(1),
r(2),  ... , r(|V(G)|).   For each appropriate  j  build  a
beam  b(j)  in  the  following  fashion.   The beam  has
north/south extent |E(G)|+1.  It has east/west extent  1/2.
It  lies  on  a  level  one unit above the  XY-plane.   In
set-theoretic terms

$$b(j) = \{(x,y,z): j - 1/4 \;<=\; x \;<=\; j + 1/4,$$
$$1/2 \;<=\; y \;<=\; |E(G)| + 1/2,$$
$$z \;=\; 1\}$$

Thus the beams all lie in the first octant in XYZ space, all
run  north/south  for a distance   |E(G)| + 1,  all are  1/2
unit across, and sit side by side like north/south crossties
on a railroad(paper-thin crossties with zero up/down extent)
whose tracks run east/west.

[6]    There are  |E(G)|  green vertices of the new graph  H.
Order  them in any fashion so that you have vertices   g(1),
g(2), ..., g(|E(G)|). For each appropriate  i  build a plank
p(i)  in  the  following  fashion.  The plank  has  east/west
extent  |V(G)|+1.  It has  north/south extent  1/2.  The  y
coordinate  of its centerline is  i.  The  x  coordinates of
its  ends are 1/2 and  |V(G)| + 1/2.  It lies in the  z =  0
plane.   In set-theoretic terms

$$p(i) = \{(x,y,z):  \qquad 1/2 \;<=\; x \;<=\; |V(G)| + 1/2$$
$$i - 1/4 \;<=\; y \;<=\; i + 1/4$$
$$z \;=\; 0\}$$

Thus  the planks all lie in the first quadrant of the  z = 0
plane  (considered as an  XY-plane), all run east/west for a
distance  |V(G)| + 1, all are  1/2 unit across, and sit side
by  side like (paper thin) east/west crossties on a railroad
whose tracks run north/south.

[7]    Suppose red vertex r(j) is joined to green vertex g(i)
in the new graph H.  Then build a pole up from  z = 0  to  z
= 1  at  (x,y) = (j,i).  In set-theoretic terms let  this
pole be the set

$$P[j,i] = \{(j,i,z): 0 \;<=\; z \;<=\; 1 \}.$$

Now  things look the following way.  A narrow-gauge railroad
track  runs east/west at  altitude 1  over  a  broad-gauge
railroad  track  running north/south  at  altitude  0  (or
vice-versa).  2|E(G)| poles are erected, as described above,

32

to support the upper ties upon the lower ties. For each
upper tie there is at least one such pole. For each lower
tie there is at least one such pole. (In fact, for any given
imbedding, there may be many ties which are in contact with
more than one pole.) Now the tracks and the ground are
gently dissolved away, leaving the ties and poles. Since
they and the epoxy they were put together with are strong,
the structure is a single rigid piece (this actually tacitly
assumes that the original graph was connected. But this
assumption is operative throughout the research. In fact we
assume that G is biconnected. Unconnected graphs simply fall
apart into subproblems). This piece of sculpture is what the
output of 3cp looks like.

[8]   Print  r(1), r(2), ... , r($|V(G)|$), g(1), g(2), ...  ,
g($|E(G)|$),  and the values  of  P[j,i]  for the  appropriate
$2|E(G)|$  members of the set

   $\{[j,i]: 1 \le j \le |V(G)|, 1 \le i \le |E(G)|\}$.

## 8.3 The Multilevel Starbody Pole (3msp) Representation of Nonplanar Graphs

The multilevel starbody/pole algorithm (3msp) imbeds a graph in 3 dimensional space in a manner analogous to the classical imbedding procedure described by Wilson [WI79, pp. 22-3].

Algorithm 3MSP:

[1] Let the original graph G have $|E(G)|$ edges and $|V(G)|$ vertices. Number the edges in any manner from 1 to $|E(G)|$.

[2] Number the vertices in G in any manner from 1 to $|V(G)|$.

[3] Then begin building corresponding terraces in the imbedding structure in the following fashion. For every positive integer j no larger than $|V(G)|$ the jth terrace T(j) will lie within a level plane with z coordinate equal to j. T(j) will contain a disk D(j) of radius 1 centered on the intersection of this plane with the Z-axis. T(j) will be starshaped with respect to this disk's center.

[4] Pause in the construction of T(j).

[5] Begin construction of the poles which will join various pairs of these terraces in the following fashion. Treat the XY-plane as if it were a complex plane to make discussion of angles and distances from the origin easy. Build a "rising sun flag with $|E(G)|$ sunrays" (resembling the Japanese flag in the early 1940s) in the following fashion. The set J, which resembles the red area in the aforementioned flag, is defined to be the union of $|E(G)|+1$ sets

$$J = D + A(1) + A(2) + \ldots + A(|E(G)|)$$

where

$$A(k) = \{w : w \text{ is a complex number such that } 2k - 1 < Arg(w) / 2\pi < 2k \}$$

for every positive integer $k <= |E(G)|$. Clearly J is a connected unbounded starshaped (with respect to the origin $(x,y,z) = (0,0,0)$ of XYZ-space) set. It is a matter of elementary trigonometry to verify that A(k) contains a square Q(k) of side 1 containing a point (i.e. complex

34

number) w such that $|w| = |E(G)|$. The square $Q(k)$ has sides parallel to the X-axis and to the Y-axis. Hence there is a point $w(k)$ belonging to $Q(k)$ with the property that both its x coordinate and its y coordinate are integers. Obviously

$$|E(G)| - 2 < |w(k)| < |E(G)| + 2.$$

For any nontrivial graph G we have the inequality

$$1 < |E(G)| - 2$$

A moment's reflection will show that this ostensibly existential argument has an algorithmic version. In each "sunbeam" of J (i.e. in each $A(k)$ outside the disk D) we thus have a point with integer coordinates at a distance from the origin of less than $|E(G)| + 2$. And we have an algorithmic way of finding this point $w(k) = (x[k], y[k], 0)$. The pole $P(k)$ will then be an upright line segment in XYZ-space. Its vertical extent will be determined by the two terraces it connects. This means that $P(k)$ must correspond to some edge $E(k)$ of the original graph G. This edge must join two vertices of G. The corresponding two terraces on the structure we are building will be called $T(p)$ and $T(q)$ where, without loss of generality, $p < q$. The vertical extent of $P(k)$ is thus known. $p \le z \le q$ for every $p = (x, y, z)$ belonging to $P(k)$. Moreover every height in this interval is found in $P(k)$. And its "footprint" will be $w(k)$, i.e. for every point $p = (a, b, c)$ belonging to $P(k)$ it will be true that $a = x[k]$, and that $b = y[k]$.

[6]     Now return to the building of terrace $T(j)$. A certain number of edges will be incident upon the vertex in G which corresponds to $T(j)$. So the poles corresponding to those edges must meet $T(j)$. Moreover all other poles must miss $T(j)$. This is easily done. Suppose the poles which must meet $T(j)$ are

$$P(k[1]), P(k[2]), \ldots, P(k[m(j)]).$$

Then we attach to the unit disk at altitude j exactly $m(j)$ "petals" which resemble truncated versions of

$$A(k[1]), A(k[2]), \ldots, A(k[m(j)])$$

elevated from altitude 0 to altitude j in XYZ-space. A petal L(j,r) is of the form

$$L(j,r) = \{(x,y,j): (x,y,0) \text{ belongs to } A(r) \text{ and } x{**}2 + y{**}2 <= (|E(G)|+2){**}2\}$$

Now we can define T(j) as the union of m(j) + 1 sets.

$$T(j) = D(j) + L(j,1) + L(j,2) + \ldots L(j,m(j)).$$

[7]    At this point we have completed building the terraces and the poles. We define the drawing as the union of the terraces and the poles. It resembles a bunch of copies of the same daisy stacked up at unit distances above one another.    They are oriented in the same fashion so that somebody looking down at the stack from above would see only the top daisy. Then various petals are torn off each daisy. Following this poles are erected on some petals to petals they can see directly above them.    This description does not do justice to the role of integer coordinates. The top and the bottom of every pole is a triple (x,y,z) of integers. The volume occupied by this drawing is less than $|V(G)|*(|E(G)|+2){**}2$.    Because of the remarks on integer coordinates, there is no problem of scale.

## 8.4    The Multilevel Convexbody Pole Representation

The multilevel starbody/pole algorithm (3msp) leaves something to be desired visually. The terraces in question look somewhat like tattered daisies, each one having a round center and a few petals. None of them will be convex.    The typical box shapes in conventional flowcharts are more often than not convex (e.g. rectangles, circles, lozenges) because they are viewed as little cards, on each of which will be written a brief description of an operation. Can we write an algorithm which produces convex terraces instead of starshaped terraces?    Yes.    Can we guarantee that the multistoried structure so produced occupies no more space than the output of 3msp does (i.e. can we still get a $|V(G)|*(|E(G)|+2){**}2$ estimate)?    Yes.    Do we pay a price for this more desirable outcome?    Yes. But an acceptable one. The proof of correctness of the new algorithm, which we shall call multilevel convexbody/pole (3mcp), is much less elementary than the foregoing proof given for 3msp.    In consequence, we will only give an outline of the 3mcp algorithm.    The interested reader who is conversant with a little analytic number theory can easily fill in the details.

The basic idea behind 3mcp is much like that behind 3msp. Again there are $|V(G)|$ terraces gotten by chopping parts off a standard "terrace floor plan". The outline goes as follows.

[1] Number the edges in any way from 1 to $|E(G)|$.

[2] Number the vertices in any way from 1 to $|V(G)|$.

[3] Build the standard terrace floor plan as a convex polygonal area in the following manner. Confine your attention to the part of the plane consisting of those points (we will regard this plane as being a complex plane for ease of geometric description) z satisfying the inequality

$$0 \leq Arg(z) < pi/4.$$

Now we look for a set G of Gaussian integers (i.e. points of the form $z = x + iy$, where x and y are integers) with several properties. It must contain more than $|V(G)|/8$ points, all of which lie in the second quadrant. No two points in G can have the same argument. If z belongs to G, and if w is a Gaussian integer such that $Arg(w) = Arg(z)$, then $|w| \geq |z|$. It must not contain $i - 1$. It must consist of small numbers in a certain well defined sense, which we will simply suggest by an example.

[3.5] The example in [3]. Imagine that $|V(G)| = 77$. The set G must have at least 10 members, since $77/8 = 9.625$. If we take

$$z[1] = 6i - 1$$
$$z[2] = 5i - 1$$
$$z[3] = 4i - 1$$
$$z[4] = 3i - 1$$
$$z[5] = 5i - 2$$
$$z[6] = 2i - 1$$
$$z[7] = 5i - 3$$
$$z[8] = 3i - 2$$
$$z[9] = 4i - 3$$
$$z[10] = 5i - 4$$

we have the desired kind of G, with its members arranged in order of increasing argument (i.e. points such the slopes of the segments joining them to $0 + i0$ are $-6,-5,-4,\ldots,-5/4$). The reader can easily see in what sense they are the smallest such points, and how they are related to the Euler

37

totient function phi. Now we build the cumulation sequence.

The cumulation sequence is defined by setting

$$c[1] = z[1] = 6i - 1$$
$$c[2] = c[1] + z[2] = 11i - 2$$
$$c[3] = c[2] + z[3] = 15i - 3$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$c[10] = c[9] + z[10] = 42i - 19$$

With this information at our disposal we form a vertex sequence

$$v[0] = 0i + 61$$
$$v[1] = v[0] + c[1] = 6i + 60$$
$$v[2] = v[0] + c[2] = 11i + 59$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$v[10] = v[0] + c[10] = 42i + 42$$

Forget about a complex structure on the plane now. Take the points

$$v[0] = (61,0)$$
$$v[1] = (60,6)$$
$$v[2] = (59,11)$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$v[9] = (38,37)$$
$$v[10] = (42,42)$$

in the XY-plane and form

$$v[11] = (37,38)$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$v[19] = (6,60)$$
$$v[20] = (0,61)$$

by reflection in the line y = x. Next we can reflect these 21 points in the Y-axis to get

38

```
                    v[21] = (-6,60)
                    v[22] = (-11,59)
                        .
                        .
                        .
                    v[40] = (-61,0).
```

Finally we reflect these 41 points in the X-axis to get

```
                    v[41] = (-60,-6)
                        .
                        .
                        .
                    v[79] = (60,-6)
```

At this point we form the convex hull of these 80 points. It
is a convex polygon with 80 vertices v(0),v(1),...,v(79) and
80 sides, and every vertex v(i) is an extreme point of this
convex set.

Thus, in this example of representing an arbitrary graph
with with 77 vertices, we constructed a polygon with 80
sides (the smallest multiple of 8 exceeding 77). This
polygon P has diameter equal to 120, and therefore area less
than 14400. The symmetry group of this polygon is either
D[4], the 8-member dihedral group of symmetries of the
square, or a group which has D[4] as a subgroup. The reason
for this is in the way P was constructed. The whole idea
was to get a lot of short sides with (of course) different
slopes and endpoints belonging to the integer lattice of the
XY-plane. These slopes were all to be less than -1. They
would be assembled to make the "east to northeast eighth" of
P by joining a side with algebraically larger slope by its
lower right end to the upper left end of a side with
algebraically smaller slope. This would produce something
looking like the right half of a haystack as shown at the
top of the next page:

This configuration would be reflected in the line y = x.
Then the original configuration would be joined to its
mirror image by a short line segment of slope -1 in the
obvious manner. This completes the full description of the
Quadrant 1 portion of P. It contains more than |V(G)|/4
line segments (i.e. at least 20 in this example). This
complete upper right quarter of P, in turn, would be
reflected in the X-axis to get the whole right half of P.
This right half of P would then be reflected in the Y-axis
to produce all of P. It is clear that the area of P is less
than $80^{**}3$.

[4] We now have a standard terrace floor plan. It is a
convex polygon with more than |V(G)| edges. We note that
each endpoint of each edge of this polygon is an extreme
point of the convex set K consisting of P, together with its
interior. The area of K (or of P, if you prefer) is less
than $|V(G)|^{**}3$. This follows from elementary analytic
number theoretic considerations [LE56, pp. 120-121] of the
average order of magnitude of the Euler totient function
phi. For each vertex v of G we customize the terrace
corresponding to v in a standard way we pick, once for all,
two extreme points of K which will not correspond to any
edge. Then we build a one to one correspondence between the
edges of G and some of the other extreme points of K. Now
take the terrace corresponding to the vertex v of G. On it,
locate all the extreme points corresponding to edges of G
which are incident on v. In addition to these edges(there
must be at least one, since G is biconnected) locate the two
extreme points which do not correspond to any edge of G. Now
form the convex hull of all the points you have just
located. This convex hull, which we shall call H(v) is
nondegenerate (i.e. has positive area) since it contains the

40

three noncollinear points mentioned immediately above. The
terrace T(v) corresponding to the vertex v of G is the set

$$T(v) = \{(x,y,h(v)): (x,y) \text{ belongs to } H(v)\}$$

where h is any numbering of the vertices of G.

[5]     Now    it  is  possible  to  complete  the    structure
corresponding  to  the graph G.  This output from the  3mcp
algorithm  is  a subset  of XYZ-space built as follows.    It
contains  the terrace T(v) for  every vertex v of G.    These
terraces are of course subsets of level planes, are pairwise
disjoint,  and are stacked up vertically above one  another.
The output structure also contains one pole for each edge of
G.    The  pole L(e)  corresponding  to edge e of G  has  the
obvious  up/down  extent.  There  are exactly  two  terraces
corresponding to the two vertices of G which are incident on
e.   Call these terraces T(v) and T(W). The up/down extent of
L(e)  lies between the  heights H(v) and h(w).  Clearly  the
pole L(e) misses all terraces other than T(v) and T(w).

This  ends  the  construction  of the output of  3mcp.    The
heights  of  the  terraces  are  pairwise  unequal  positive
integers. The "footprints" of the poles are pairwise unequal
integer  lattice points in  the XY-plane.  In fact they  all
lie  on the boundary of  K, among its extreme points.    Thus
the  algorithm  vercomes  the  problem of scale.  The  output
structure, i.e. the drawing of G, is |V(G)| stories high and
its    footprint  is  contained  in  a  convex  set  of  area
|E(G)|**3.    Thus  we have  a |V(G)|*|E(G)|**3 bound on  the
size of the "solid flow diagram".

41

## 8.5 The Basketweave (2.5b) Representation of Nonplanar Graphs

The basketweave algorithm (2.5b) draws a graph in a plane, but with crossovers. Its output is similar in looks and spirit to that of the jailcell algorithm (2jc). Its input is the adjacency matrix of a graph G with $|E(G)|$ edges and $|V(G)|$ vertices. It acts as follows.

[1]    Number the vertices in any way. Represent the ith vertex by a brick with horizontal extent from $x = 1/2$ to $x = |E(G)| + 1/2$, and with vertical extent from $y = i - 1/4$ to $y = i + 1/4$. Thus the brick is $|E(G)|$ by $1/2$, and its center is at height i.

[2]    Number the edges in any way. Suppose the jth edge joins vertices $i(1,j)$ and $i(2,j)$ in the graph G. Represent the jth edge by a pole whose points all have x coordinate equal to j. Assume, without loss of generality, that $i(1,j) > i(2,j)$. We want to have this jth pole join brick $i(1,j)$ to brick $i(2,j)$ but not touch any other brick. This is accomplished by having it tunnel under bricks at altitudes between $i(2,j)$ and $i(1,j)$. More on this presently.

[3]    Consider the ith brick again. It touches a collection of poles. The common x coordinate of all the points on the leftmost pole in this collection will be called $L(i)$. The common x coordinate of all the points on the rightmost pole on this collection will be called $R(i)$. We now erase part of the ith brick to get it in its final form as

$$B(i) = \{(x,y): L(i) - 1/2 <= x <= R(i) + 1/2,$$
$$i - 1/4 <= y <= i + 1/4\}.$$

The ith brick goes only as far across the page as is necessary to get it to touch all the poles corresponding to the edges which touch the vertex it represents.

[4]    Consider the jth edge again. The corresponding pole consists of points whose x coordinate is j, and whose y coordinate lies between $i(2,j) + 1/4$ and $i(1,j) - 1/4$. Consider any positive integer i such that

$$i(2,j) < i < i(1,j)$$

If there is a point of $B(i)$ whose x coordinate is equal

42

to  j  we must drive the corresponding part of the jth pole
under  the  brick  B(i).  Note  that we do not  regard  this
logically  as removing this part of the  jth pole, merely as
virtually  dropping this part to the other side of the sheet
of paper on which it is drawn.  The part of the  jth pole we
will drive under B(i) will be the segment

$$\{(j,y):  i - 5/16  \leq  y  \leq  i + 5/16\}.$$

The visual effect of this mode of drawing is like a birdseye
view  of  a bunch of  parallel roads (the bricks) which  are
elevated one level above ground.  There are, additionally, a
bunch of north/south wires (the poles) at ground level.  But
each  end of each wire rises up from ground level to  attach
to a road.

[5]    Use some tunnel-under symbol at each point where part
of  a pole is moved (virtually) from the front of the  sheet
of  the paper to the back (or from the back to the front) if
you wish.

At  this  point the drawing  is finished.  It looks  like  a
simple-minded  basketweave in which every pole tunnels under
every  brick  it  meets,  except  the  bricks  at  its  two
endpoints.  Once again there  is no problem of scale.   The
poles  have  integer  x  coordinates.  The centers  of  the
bricks  have  integer  y  coordinates.  The entire  drawing
occupies a $|V(G)| + 1$  by  $|E(G)| + 1$  rectangle.  All flows
are  vertical.  All  operation  boxes  (the  bricks)   run
horizontally  across  the  page.  The  tunnel-unders  are
obvious,  even  in  the  absence  of  a  back  line  removal
procedure and a tunnel-under symbol.  A pole tunnels under a
brick  if and only if that brick is not attached to the  top
or the bottom of that pole.

The basketweave algorithm (2.5b) is the sole point where the "extent of nonplanarity" of a graph arises as a pictorial, rather than as a merely logical, consideration in the work being reported upon. If two different drawings of the same ancestral flow diagram are both done in basketweave fashion (i.e. both consist entirely of "horizontal" bricks and vertical poles, some of which poles tunnel under bricks) we would regard one of the two as superior if it had fewer "crossovers" (places where a single pole tunnels under a single brick). Finding such a drawing, i.e. one with a minimal number of crossovers probably amounts to solving the crossing number problem [GJ79], which is NP-complete. For large graphs this is a costly procedure. As noted elsewhere in this report, system design usually proceeds in top-down fashion. Hence most flow diagrams considered have only about 20 boxes.

Most of the time 2.5b will produce a drawing with no more than 2 crossovers for graphs with at most 20 nodes. Two possibilities arise. The graph in question may be planar. In this case the jailcell algorithm (2jc) will produce a planar jailcell drawing. If the graph in question is not planar then we can settle this case of the crossing number problem (possibly accepting a time penalty if the graph has many more than 20 vertices, which is unlikely as noted above), and see whether there are more crossovers than necessary in the drawing which is the output of 2.5b. In the unlikely event that there are too many crossovers, it remains to remove the superfluous ones. An algorithm to do this is a Phase 2 goal. A simpleminded and expensive algorithm would be simply to generate all possible 2.5b representations of the graph and look through the collection for one with the minimal number of crossovers.

Implicit in the foregoing is the obvious utility of a basketweave drawing of a flow diagram. Both the jailcell and the basketweave approaches offer a common solution to the problem of spaghetti flow diagrams, undisciplined unstructured messes with sinuous merging crossing flows, and boxes with no consistent orientation to provide appropriate visual cues.

The solution is "horizontal" bricks replacing all boxes of whatever shape. The operation is written on the brick. The flows are, without exception, vertical poles. Hence there is no possibility that flows can merge or cross. Flow begins

at an obvious source and moves systolically, with downward flow and occasional upward ebb, and systematically, tending ever downward toward an obvious sink. No two operation bricks are at the same height. Hence there is a natural labelling scheme. Label the operation written on a brick with the height of that brick. Figure 3.2 of Appendix E resembles the output of 2.5b for a nonplanar flow diagram, and Figure 4.3 resembles the output of 2.5b for a planar flow diagram. Observe how much more orderly and readable it is than the equivalent Figure 4.1

Flow diagrams are the natural high level language in which system design should be expressed. The closest competitor, written English, is woefully inadequate because of its ambiguity and its inability to give natural and brief expression to notions of adjacency, previous state, and next state. The fact is, however, that they are seldom used in the early stages of system design. Indeed they are often absent from all stages of system design.

Why is such a natural approach, free from all dependence on a choice of programming language, so widely eschewed? Two parts of the answer are spaghetti and scale. Before the algorithmic approaches reported on here, humans and their drawing aids simply tended to produce bad drawings of the flow diagrams implicit in a rough overview of the design of any fairly complex system. After a few boxes have been drawn the problem of scale occurs, not as a mathematical abstraction but as the need to put an operation box into an unanticipatedly awkward spatial relationship with a bunch of already drawn operation boxes. The result is an unnatural placement of the new operation box and a spaghetti set of flows between it and the already existing boxes. Once the initial flaw appears the problem ramifies rapidly. The problem is reminiscent of VLSI design, in its obvious need for algorithmic remedies. And the solutions reported upon here are similar in spirit, though considerably different in detail, from those solutions.

In summary, 2jc and 2.5b do more than enable us to draw readable flow diagrams. They enable us for the first time to draw compact flow diagrams, and these compact flow diagrams (amazingly) have the added advantage that they are more readable than all but the few best flow diagrams produced in conventional ways.

In particular, algorithm 2jc enables us to draw planar graphs compactly and without the problem of scale which has

plagued all previous algorithms for imbedding planar graphs of arbitrary degree, and algorithm 2.5b enables us to draw nonplanar graphs in a way which approximately minimizes the number of crossings and which therefore indicates the "amount" and "location" of unstructuredness of flowcharts based on those graphs.

It is time to reexamine the whole matter of whether and when to use the power of flowcharts in the design of large, complicated systems.

The objections of Dijkstra [DI68] and others to the use of the GOTO statement is based on the fact that it is hard to read and understand "spaghetti code"; that is, code in which a single function is distributed throughout a large program but logically connected by unconditional branches. What Dijkstra realized was that it is easier to write spaghetti code using the powerful and unconstrained GOTO statement than it is to write such code in a more constrained language like Pascal. This in itself, however, is not reason enough to banish the unconditional branch from the programmer's repertoire of tools; the use of the GOTO statement does not always result in spaghetti code. In fact, as has been demonstrated above, there are situations in which unconditional branches are desirable or even inevitable.

Some programming language designers have come to the conclusion that the spaghetti code problem is an artifact not of the unconditional branch, but rather of the "imperative programming paradigm" which is based on the Von Neumann model of computation and which underlies most currently popular programming languages (LISP being the only common exception.) Over the past several years a variety of new programming paradigms have been developed which allow a programmer to use the power of unconditional branching (often implicitly) while avoiding the pitfalls of spaghetti code.

Ada, the DoD's language of choice, incorporates some ideas from one of these new paradigms, namely the package paradigm. In Ada, it is possible to write a program as a collection of packages, each of which knows only about the interfaces (not the internal implementation details) of the other packages in the system. This invisibility of the internal structure and state of packages is called information hiding. These packages then communicate with one another to accomplish the work of the program.

The package paradigm is quite similar in concept to several other recently developed programming paradigms, notably the process, dataflow, and object-oriented paradigms. In each of these paradigms, an environment consists of a population of entities (objects, dataflow nodes, or processes), whose internal details are invisible to the other entities in the population. A program in one of these paradigms consists basically of a connection map, which specifies what pairs of entities can communicate with one another, and a sequence of

47

initial communications, which are sent to specified entities in the environment. These intial communications trigger other communications among the entities in the environment, and eventually a communication or sequence of communications is transmitted back to the "outside world" as the output of the program.

The rationale behind these paradigms is that if programs can be broken up into a number of self-contained entities, the entities can be made simple enough so that they are easy to understand, write, debug, and maintain. Since no entity can "see" the internal details of another, no entity can be written in such a way that it depends on another's implementation, which greatly reduces the probability that a change in one part of a program will cause unexpected errors in another, unrelated part.

The primary difficulty with these paradigms is that while it is often easy to write the code for the individual entities, it is often quite hard to visualize the overall structure of the program, with the result that integration of the entities into a single large program presents substantial difficulties. Since each entity generally has an interface to the environment which contains a number of different input possibilities and a number of different behaviors for each input possibility, the interactions of even a small population of entities can become quite complex.

As an example of these difficulties in visualizing the global behavior of a system (in this case an object-oriented system), we present appendix C, which contains a Smalltalk program. It is not within the scope of this report to provide a Smalltalk tutorial, but we will mention a few things to make the example understandable. Each entity in Smalltalk is an object. Each object is a member of a class. Each class defines the types of messages which objects of that class can respond to. A message has a receiver (which is an object), a selector, and a parameter list. So for example, the message "zensort step3150:input." has receiver "zensort" (zensort is an object), selector "step3150:", and parameter list "input". When an object (zensort, in the case of our example) receives a message, it looks at its class definition and determines whether it has a method to implement that message. If not, it creates an error. If so, it executes the method. A method is simply a sequence of communcations to be sent. In our example, since "zensort" is of class "Shuffler", it looks in its class definition and finds a method called "step3150:" (the first

48

method), so it sends the messages associated with that method (starting with "(aList at:1) > (aList at:2)"; the |temp| notation indicates that the method creates a temporary variable called "temp".)

The environment for this program contains only one object, of type "Shuffler". This object can receive messages of 4 types (step3150, step3350, step3550, and step3650). Each message must have a list as its parameter. The response to each message is quite simple: the object generally compares two of the values in the list, then (depending on the result of the comparison) sometimes swaps two values in the list, and finally returns the (possibly modified) list to the object which sent the message (which in this case is always itself.) The program itself consists of the initial communications which appear at the beginning (starting with "input := Array new:3" and ending with "zensort step3150:input.") It is evident that an object of class "Shuffler" performs some kind of sorting or permutation procedure; it may not be so obvious at first glance that it actually implements the K(3,3) Zensort algorithm. Nevertheless, "Shuffler" does implement K(3,3) Zensort, and appendix C is an example of a system with only one object whose global algorithm is nonplanar. At the same time, appendix C illustrates how a language like Smalltalk (which is a standard object-oriented language) can replace the hard-to-understand GOTO statement with a simpler primitive (in this case, message passing), and produce programs which are easier to read (for one familiar with the language) than a BASIC or FORTRAN program implementing the same algorithm would be to an experienced BASIC or FORTRAN programmer. This program was written and tested in Digitalk Smalltalk-V.

A number of references dealing with the object-oriented, dataflow, and process paradigms appear in the bibliography of this report.

49

## 11.0       What are Nonplanar Algorithms Good For?

As we noted in the previous section, nonplanar algorithms arise naturally as descriptions of the global behavior of complex programs written in object-oriented, dataflow, or process-paradigm languages. One of the barriers to the widespread adoption of these languages is that, while the behavior of individual entities is easily specified and understood, the global behavior of the system is often forbiddingly complex. None of the standard representation techniques has, at the present time, proved equal to the task of specifying the global structure of large programs in such languages. This is important because a good, understandable representation of the global behavior of a program in such a language could greatly simplify the system integration task.

A particular class of object-oriented systems are called cellular automata. These systems have been used to describe biological phenomena as well as high-energy physics, and are the objects of increasing amounts of study. It is possible that nonplanar flowcharts will prove valuable in the description of the complex global behavior of such automata.

Finally, nonplanar flowcharts should, as indicated above, be applicable to the description of large imperative-paradigm programs which incorporate error control in an otherwise structured framework.

All of these application areas are open for investigation in a phase-2 effort.

11.0 Appendix A: The Objectives of the Reported Research Effort as contained on pages 2 and 3 of the award/contract document.

# PART I - THE SCHEDULE

## SECTION B - SUPPLIES OR SERVICES AN PRICS/COSTS

0001   RESEARCH

The contractor shall furnish the level of effort specified in Section F, together with all related services, facilities, supplies and materials needed to conduct the research and prepare the reports described below.  The research shall be conducted during the period specified in Section F.

0001AA

   a.  Using appropriate local information, to produce a global "solid" flow diagram (i.e. a very special kind of representation of a digraph in 3-space);

   b.  Using appropriate local information, to determine whether there is a planar equivalent of this solid flow diagram;

   c.  Producing an optimal drawing on paper of a flow diagram if that diagram is nonplanar.  An optimal drawing is one which has a minimal number of crossovers;

   d.  Moving from local information to pictorial representations (3-dimensional if necessary) of the flow diagram which are extremal in any one of a variety of ways.  One type of extremality would be a display which would suggest a maximally parallel implementation which would buy speed at the cost of using several processors.  Such representation techniques may help in understanding a program written from, or corresponding closely to, such a flow diagram.

   e.  Examining nonplanar flow diagrams in their own right with a view to understanding the kinds of programs which correspond to them.  During the last twenty years we have learned a lot about how to read and write "good" structured (i.e. planar) programs.  Now that we know that nonplanar structures exist, and cannot be made to go away, it is time to seek for a comparable improvement in our ability to understand snarls and their relationship to the larger structures they reside in;

   f.  Examining "batching", an analog of pipelining which is appropriate to snarls and which promises many fold speedup of programs involving snarls;

   g.  Ascertaining whether some biological systems act in ways which seem naturally to correspond to code with snarls, and attempt to give examples of code with snarls which is in some way superior to planar code for carrying out the same task;

h. Examining whether unbounded cellular automata, which are more general then Turing machines, can be used as a basis for "3-dimensional" computer languages which need no GOTOs. In this way we might be able to exercise GOTO after all. But the process would be general and scientific, rather than merely moralistic ("The writing of structured programs is a mark a good taste. So write them!") or improperly based (on unjustified planarity assumptions about flow diagrams).

0002   REPORTS

Reports identified below shall be prepared in accordance with Exhibit A to this contract and delivered in accordance with Section F.

0002AA   FINAL REPORT

Other reports which are or may be required under this contract are identified in Sections H and I.

See Section H, Paragraph 3 for pricing information.


## SECTION D - PACKAGING AND MARKING


### 1. PACKAGING AND MARKING

Pack in accordance with standard commercial practices and mark for the addressee shown in paragraph 2 of Section G.


## SECTION E - INSPECTION AND ACCEPTANCE


### 1. INSPECTION  AND ACCEPTANCE

Inspection and acceptance of all deliverable items called for under this contract will be performed by AFOSR, Bolling AFB DC. Official notification of acceptance shall be made by the Air Force Office of Scientific Research, Directorate of Contracts.


## SECTION F - DELIVERIES OR PERFORMANCE


### 1. PERIOD OF PERFORMANCE AND DELIVERY

The supplies and services described in Section B shall be delivered or performed during the following period:

0001AA   15 Aug 86   through 14 Feb 87.

0002AA   FINAL REPORT DUE 14 Apr 87

YLYK Ltd
F49620-86-C-0103

Presented Below is a structured (Pascal-like) version of the
K(3,3) Zensort algorithm, implemented in BASIC. It could
not be implemented conveniently in Pascal, since Pascal does
not allow branches (GOTO's) into the range of loops from
outside the loop branched into. note that there is only one
branch (GOTO) statement in this implementation, but it does
branch into a loop at statement 3550.

```
REM ***********************************
REM *
REM * THIS IS A MICROSOFT (R) QUICKBASIC (C)
REM * PROGRAM WHICH IMPLEMENTS THE K(3,3)
REM * ZENSORT ALGORITHM IN A STRUCTURED WAY.
REM * NOTE THAT THIS PROGRAM WOULD NOT WORK
REM * IN PASCAL, SINCE THE BRANCH TO 3550 IS
REM * A BRANCH OUT OF ONE BLOCK AND INTO ANOTHER,
REM * WHICH PASCAL CONSIDERS ILLEGAL.
REM *
REM ***********************************

INPUT "Enter A, B, and C separated by commas"; A, B, C

     QUITFLAG = 0    ' 0 represents FALSE; -1 represents
TRUE.

     WHILE (NOT QUITFLAG)
       ALEBFLAG = 0

       IF (A <= B) THEN
3550       IF (B <= C) THEN QUITFLAG = -1 ELSE ALEBFLAG = -1
       END IF

       IF (NOT QUITFLAG) THEN
         IF ((NOT ALEBFLAG) OR (A > C)) THEN
           TEMP = A : A = C : C = TEMP  ' Switch A and C
           IF (A <= B) THEN GOTO 3550
         END IF

         TEMP = B : B = C : C = TEMP     ' Switch B and C
       END IF

     WEND
     PRINT
     PRINT "The numbers in sorted order are: "; A, B, C
```

Presented Below is an unstructured (FORTRAN-like, without loop statements) implementation of the K(3,3) Zensort Algorithm.

```
REM ******************************************
REM *
REM * THIS PROGRAM IS AN UNSTRUCTURED IMPLEMENTATION
REM * IN MICROSOFT (R) QUICKBASIC (C) OF THE K(3,3)
REM * ZENSORT NON-PLANAR SORTING ALGORITHM.
REM *
REM ******************************************
REM
REM ******************************************
REM *
REM * FIRST WE READ IN THE VALUES OF THE THREE
REM * NUMBERS (A, B, AND C)
REM *
REM ******************************************

      INPUT;  "Enter A, B, and C separated by commas: ",  A,
B, C

3150  IF (A <= B) THEN GOTO 3550 ELSE GOTO 3260
3550  IF (B > C) THEN GOTO 3650

REM ******************************************
REM *
REM * IF WE GET HERE, THE NUMBERS ARE IN SORTED
REM * ORDER, SO WE PRINT THEM OUT IN ORDER.
REM *
REM ******************************************

3670  PRINT
      PRINT "Numbers in increasing order are: "; A, B, C
      END
```

```
REM ********************************************
REM *
REM * IF WE GET HERE, WE NEED TO SWITCH A AND C.
REM *
REM ********************************************

3260  TEMP = A
      A = C
      C = TEMP


3350  IF (A <= B) THEN GOTO 3550 ELSE GOTO 3460
3650  IF (A > C) THEN GOTO 3260

REM ********************************************
REM *
REM * IF WE GET HERE, WE NEED TO SWITCH B AND C.
REM *
REM ********************************************

3460  TEMP = B
      B = C
      C = TEMP

      GOTO 3150
```

"Note that our word-processor lacks the up-arrow
character which Smalltalk-80 uses to indicate the
function which returns a value to the calling
object.  We substitute the word Return for this
character in the program below."

"HERE IS A SMALLTALK PROGRAM TO PERFORM THE K(3,3)
ZENSORT PROGRAM.  IT MAKES USE OF AN OBJECT CALLED
zensort OF CLASS Shuffler.  THE DEFINITION OF THE
METHODS FOR CLASS Shuffler APPEARS BELOW."

```
| input zensort |                "temporary variables"
input := Array new:3.
input at:1 put:3.
input at:2 put:1.
input at:3 put:2.
zensort := Shuffler new.
zensort step3150:input.
```

"HERE ARE THE METHODS FOR THE Shuffler CLASS"

```
step3150: aList

     "Perform boxes 3150 and 3260 of K(3,3)
      Zensort algorithm"

   | temp |

(aList at:1) > (aList at:2)
   ifTrue: [temp := (aList at:1).
           aList at:1 put:(aList at:3).
           aList at:3 put:temp.
           Return(self step3350: aList)]
   ifFalse:[Return(self step3650: aList)].
```

```
step3350: aList

    "Perform steps 3350 and 3460 of K(3,3)
     Zensort algorithm"

    | temp |

(aList at:1) > (aList at:2)
  ifTrue: [temp := (aList at:2).
           aList at:2 put:(aList at:3).
           aList at:3 put:temp.
           Return(self step3150:aList)]
  ifFalse:[Return(self step3650:aList)].

step3550: aList

    "Performs steps 3550, 3260, and 3460 of the
     K(3,3) Zensort algorithm (note that 3260
     and 3460 are also performed by other methods)"

    | temp |

(aList at:1) > (aList at:3)
  ifTrue: [temp := (aList at:1).
           aList at:1 put:(aList at:3).
           aList at:3 put:temp.
           Return(self step3350:aList)]
  ifFalse:[temp := (aList at:2).
           aList at:2 put:(aList at:3).
           aList at:3 put:temp.
           Return(self step3150:aList)].

step3650: aList

    "Performs steps 3650 and 3670 of K(3,3)
     Zensort algorithm"


(aList at:2) > (aList at:3)
  ifTrue: [Return(self step3550:aList)]
  ifFalse:[Return aList].
```
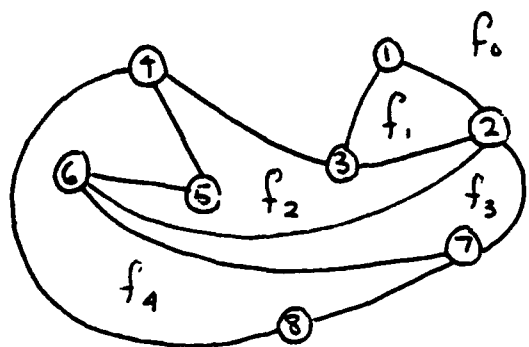
# 14.0    Appendix D:   An Example of the Operation of the Jailcell Algorithm
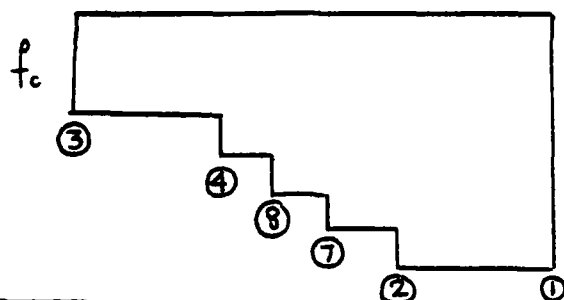
THE ORIGINAL GRAPH.

WE CHOOSE $f_0$ TO BE THE EXTERIOR VERTEX.

WE CHOOSE EDGE $(1,3)$ TO BE DRAWN FIRST.

THE IMBEDDING-ORDER LIST IS EMPTY AT THE START.

AFTER EMBEDDING EXTERIOR FACE. THERE IS ONE "REGION" (INTERIOR OF THE FIGURE). IMBEDDING-ORDER LIST IS:
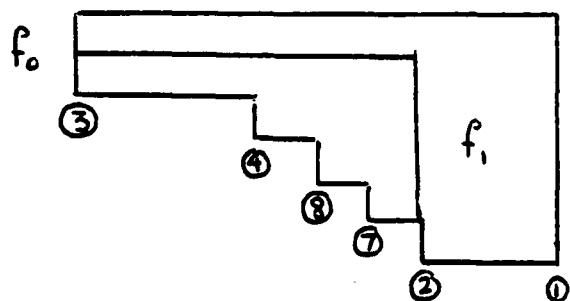
$$(f_., f_2, f_4, f_3)$$

AFTER IMBEDDING $f_1$.

THERE IS ONE REGION.

IMBEDDING-ORDER LIST IS: $(f_2, f_4, f_3)$. ONLY ONE PATH WAS DRAWN TO IMBED $f_1$.
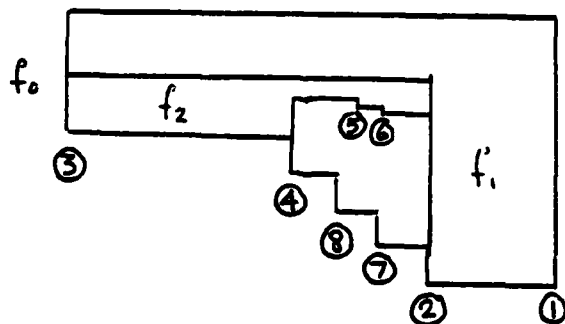
AFTER IMBEDDING $f_2$. ONE PATH WAS DRAWN. THERE IS ONE REGION. IMBEDDING-ORDER LIST IS $(f_4, f_3)$.
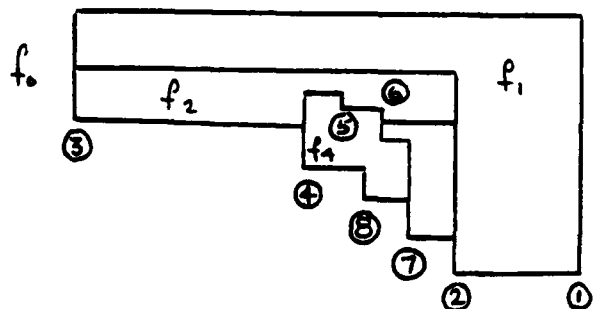
AFTER IMBEDDING $f_3$. ONE PATH WAS DRAWN. THERE IS ONE REGION. IMBEDDING-ORDER LIST IS

$$(f_3)$$

AFTER IMBEDDING $f_4$. NO PATHS WERE DRAWN THERE ARE NO REGIONS. IMBEDDING-ORDER LIST IS EMPTY.

15.0 Appendix E: The Technical Portion of the Proposal Which Led
to This Contract

# DEFENSE SMALL BUSINESS INNOVATION RESEARCH (SBIR) PROGRAM

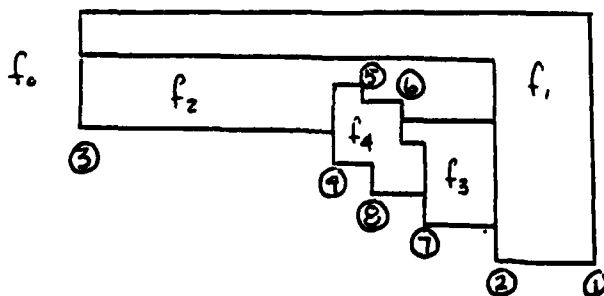Topic Number: AF86-12  ☐ Army   ☐ Navy   ☒ Air Force   ☐ DARPA   ☐ DNA
☐ SDIO   ☐ DCA   ☐ DMA

Title Proposed by Firm: Reduction of flow diagrams to unfolded form modulo snarls

Submitted By:   Firm ___ YLYK Ltd.

Address ___ 2440 Stone

City ___ Ann Arbor ___ State __ MI __ Zip Code __ 48105

Submitted To: (Activity identified with the topic) ___ AFOSR/XOT

Building 410, Room A113

Address ___ Bolling AFB

City ___ Washington ___ State __ DC __ Zip Code __ 20332-5000

## Small Business Certification:

The above firm certifies it is a small business firm and meets the definition stated in the Small Business Act 15 U.S.C. 631 and in the Definition Section of the Program Solicitation.

"The above firm certifies that it ___ does __X__ does not qualify as a minority or disadvantaged small business as defined in the Definition Section of the Program Announcement."

The above firm certifies that it qualifies as a woman-owned small business firm :
Yes __X__ No ___

Disclosure permission statement as follows:

All data on Appendix A is releasable information. All data on Appendix B, for an awarded contract, is also releasable.

"Will you permit the Government to disclose the information on Appendix B, if your proposal does not result in an award, to any party that may be interested in contacting you for further information or possible investment?
Yes __X_ No ___."

Number of employees including all affiliates (average for preceding 12 months): ___ 2 ___

Proposed Cost (Phase I): $ 49,973

Proposed Duration: __6__ months (not to exceed six months).

| Project Manager/Principal Investigator | Corporate Official (Business) |
|---|---|
| Name ___ Bob Blakley | Name ___ Bob Blakley |
| Title ___ President, YLYK Ltd. | Title ___ President, YLYK Ltd. |
| Signature | Signature |
| Date ___ 21 January 1986 | Date ___ 21 January 1986 |
| Telephone ___ 313-994-1291 | Telephone ___ 313-994-1291 |

For any purpose other than to evaluate the proposal, this data except Appendix A and B shall not be disclosed outside the Government and shall not be duplicated, use, or disclosed in whole or in part, provided that if a contract is awarded to this proposer as a result of or in connection with the submission of this data, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the funding agreement. This restriction does not limit the Government's right to use information contained in the data if it is obtained from another source without restriction. The data subject to this restriction is contained in page(s) ___ of this proposal. Failure to fill in all appropriate spaces may cause your proposal to be disqualified.

# SMALL BUSINESS INNOVATION RESEARCH PROGRAM
## PHASE 1 — FY 1986
## PROJECT SUMMARY

Topic No. __AF86-12__          Military Department/Agency __Air Force__

---

**Name and Address of Proposing Small Business Firm**

YLYK  Ltd.
2440 Stone
Ann Arbor, Michigan   48105

**Name and Title of Principal Investigator**

Bob Blakley
President, YLYK  Ltd.

**Title Proposed by Small Business Firm**

Reduction of flow diagrams to unfolded form modulo snarls

---

**Technical Abstract**  (Limit your abstract to 200 words with no classified or proprietary information/data.)

This proposal gives what may be the first examples of nonplanar flow diagrams, i.e.
flow diagrams describing code which is intrinsically incapable of having all
unconditional branches removed.  Some reflection on the significance of these examples
leads to a realization that a variety of desirable goals in software engineering are
now both desirable and within reach.  It is proposed to seek to attain several goals
including: developing an algorithm which takes purely local information on
relationships between parts of a program and produces a global flow diagram;
developing an algorithm for determining whether this diagram is planar in the
graph-theoretic sense; developing an algorithm which takes the solely local
information in a diagram known, on mathematical grounds, to be planar and produces a
plane drawing of it without crossovers; developing an algorithm which takes the solely
local information in a demonstrably nonplanar diagram and produces a plane drawing of
it which has a (provably) minimal number of crossovers; developing algorithms for
moving from local information to drawings which are extremely informative in regard to
some chosen aspect of a proposed program; elaborating a theory of "nonplanar" programs
comparable in power, and complementary in scope, to structured programming as a theory
of planar programs; building high performance nonplanar algorithms which, being
subject to fewer restrictions, outperform structured programs.

**Anticipated Benefits/Potential Commercial Applications of the Research or Development**

It is now clear that code which does not conform to the canons of structured
programming is not necessarily bad code.  Fully automated routines for moving a
proposed program as far as possible toward structured form promise to yield
simplifications of software design procedures.  The class of nonplanar flow diagrams
probably describes many programs and algorithms which are superior from the viewpoint
of speed, memory requirements and other desirable properties of code to their
structured (i.e. planar) opposite numbers.

---

**List a maximum of 8 Key Words that describe the Project.**

Unconditional branching, GOTO, structured programming, software engineering,
flowcharts

## 3. Identification and significance of the problem/opportunity.

The cost of producing software, to a large extent a byproduct of the unproductive labor-intensive use of skilled professionals to write it, continues to be a burden to DoD and others. The proposed research is aimed at a return to first principles to increase our understanding of the interrelationships among the parts of a program or software package. The research will make nontrivial use of the combinatorial topology of flow diagrams to produce new methodologies in the design and analysis of algorithms. We define a flow diagram to be what our intuitive notion of flowchart suggests, a digraph (i.e. directed graph [WI79, p. 9]) whose adjacency matrix [BO77, p. 173] is a 0,1 square matrix with zero trace [LA66, p. 51]. A more intuitively appealing way to state this definition of a flow diagram is as a digraph with no more than one arc from u to v (for any vertices u, v) and such that its underlying graph [WI79, p. 100; BE71, p. 6] is a graph [BE71, pp. 1-8] with no loops [BO77, p. 3].

On the one hand, the proposed research will involve a critique and a reformulation of some of the basic concepts in structured programming [DA72; JA75; MI72b; JE81; WI71; WI74] and the theory of structured flow diagrams [BE85; BO66; HU82]. This should dispel a widely held misconception of the meaning of Dijkstra's note [DI68] on unconditional branching and the Böhm-Jacopini paper [BO66] on planar algorithms. It should also help publicize the consequent need to take account of a special type of subprogram, called a _snarl_ on page 7 below, when discussing structured objects.

On the other hand, the proposed research is aimed at producing an algorithm to build a reasonable global picture of a program from a complete set of local descriptions of it. This will involve using nontrivial topological and combinatorial tools. In the best possible outcome, this research may make it possible to design maximally parallel (as well as maximally serial) renditions of any given piece of proposed software based only on this local information. It may also be possible to use the concept of snarl to search purposefully for the real trouble spots and the opportunities for speedup or parallelization in very general software design projects.

Finally, this research may lead to a formulation which allows us, using cellular automata rather than Turing machines, to do what many people wrongly believe Böhm and Jacopini [BO66] and Dijkstra [DI68] did, namely read unconditional branching procedures out of the universe of civilized discourse in computer science.

### 3a. There are nonplanar flow diagrams.

We must depart from the usual proposal format for a considerable digression at this point. The topic is nonplanar flow diagrams and code with unremovable GOTOs.

---

*Footnote: All entries in square brackets refer to the bibliographic citations list contained in Subsection 6b below.

To get down to specifics, it is necessary to start by describing what the state of affairs is not. A tutorial [JE81] in IEEE Computer, states:

> ... Böhm and Jacopini ... demonstrated that three basic
> control structures were sufficient to express any
> flowchartable program logic. These basic constructs ...
> include a sequence mechanism, a selection mechanism, and an
> iteration mechanism.

and more recently a paper [BE85] in the Journal of the ACM states:

> Böhm and Jacopini ... showed that any algorithm can be
> implemented as a D-chart.

A superficial reading of these comments, or of the Böhm/Jacopini [BO66] paper, might leave a misconception. It is true that there is a very large class of tasks which Turing machines can carry out, and with the property that each task belonging to that class can be carried out by an algorithm expressible by a planar flow diagram (i.e. a flow diagram whose underlying graph is imbeddable in the plane [WI79, p. 59]). It is not true that every algorithm to perform each such a task can be described by a planar flow diagram. Nor does Dijkstra make this strong claim in the last paragraph of [DI68], despite an aversion [DI76] to GOTO, shared with Hoare [DA72], Wirth [WI71; WI74] and others [KE78], so strong that it sometimes seems they want nothing less than to drop it down the memory hole. Also there might be tasks people might want performed which cannot be carried out by any algorithm with a planar flow diagram. But settling this latter question and finding such tasks might be very difficult, involving delicate considerations related to Church's thesis [MA77, pp. 177-205; CO89, pp. 790-800].

Despite the obvious importance of the subtle distinction alluded to in the previous paragraph, nobody has bothered, heretofore, to produce a demonstrably nonplanar flow diagram. Part of the reason for this is, of course, the necessity to do a good bit of foundational spadework to justify the example, and part may be due to failure to connect the combinatorial topology of graphs [WI79, pp. 59-63] with this question.

We take this opportunity to remind the reader that a graph is planar if and only if [WI79, p. 62] it contains no subgraph which is contractible to $K(3,3)$ or $K(5)$.

It is important, however, to see just such nonplanar flow diagram examples in order to catch the spirit of the proposed research. Considerations of space make it impossible to provide all the foundational material here, but the examples $K(3,3)$ ZENSORT and $K(5)$ ZENSORT shown in Figures 3.1 and 3.2 below make the point quite clearly to anybody who chooses to study them.

There are numerous planar flow diagrams of algorithms to accomplish the task of sorting three real numbers, but $K(3,3)$ ZENSORT is nonplanar. Similarly $K(5)$ ZENSORT is an intrinsically nonplanar flow diagram describing an algorithm to sort four real numbers. To verify the correctness of $K(3,3)$ ZENSORT it suffices to see that it works on each of the 27 members of the set $\{1,2,3\}^{\{1,2,3\}}$ of lists of three members of the set $\{1,2,3\}$. Similarly $K(5)$ ZENSORT is correct if it sorts every one of the 256 lists belonging to $\{1,2,3,4\}^{\{1,2,3,4\}}$. These two rather obvious

```
                        BEGIN                    3130

                     READ (A,B,C)                3140

                     IS A ≤ B ?                   3150
                  NO              YES

            (A,B,C) IS REPLACED BY (C,B,A)        3260

                     IS  A ≤ B ?                  3350
                  NO              YES

            (A,B,C) IS REPLACED BY (A,C,B)        3460

         NO      YES
                   IS A ≤ C ?                     3550
              NO

                     IS  B ≤ C ?                  3650
                         YES

                    WRITE (A,B,C)                 3670

                        END                      3680
```
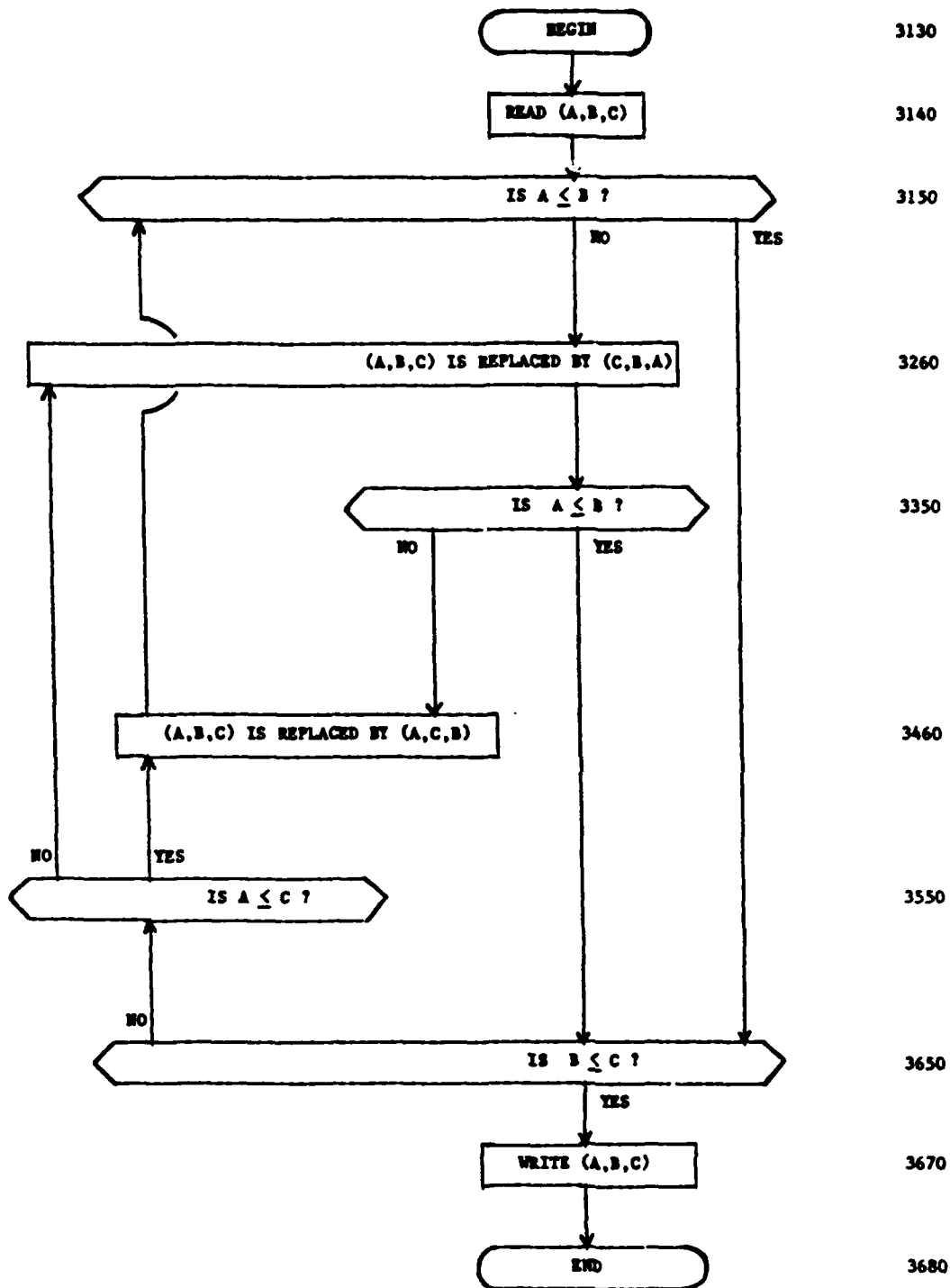
Figure 3.1

A flow diagram which defines the K(3,3) ZENSORT algorithm
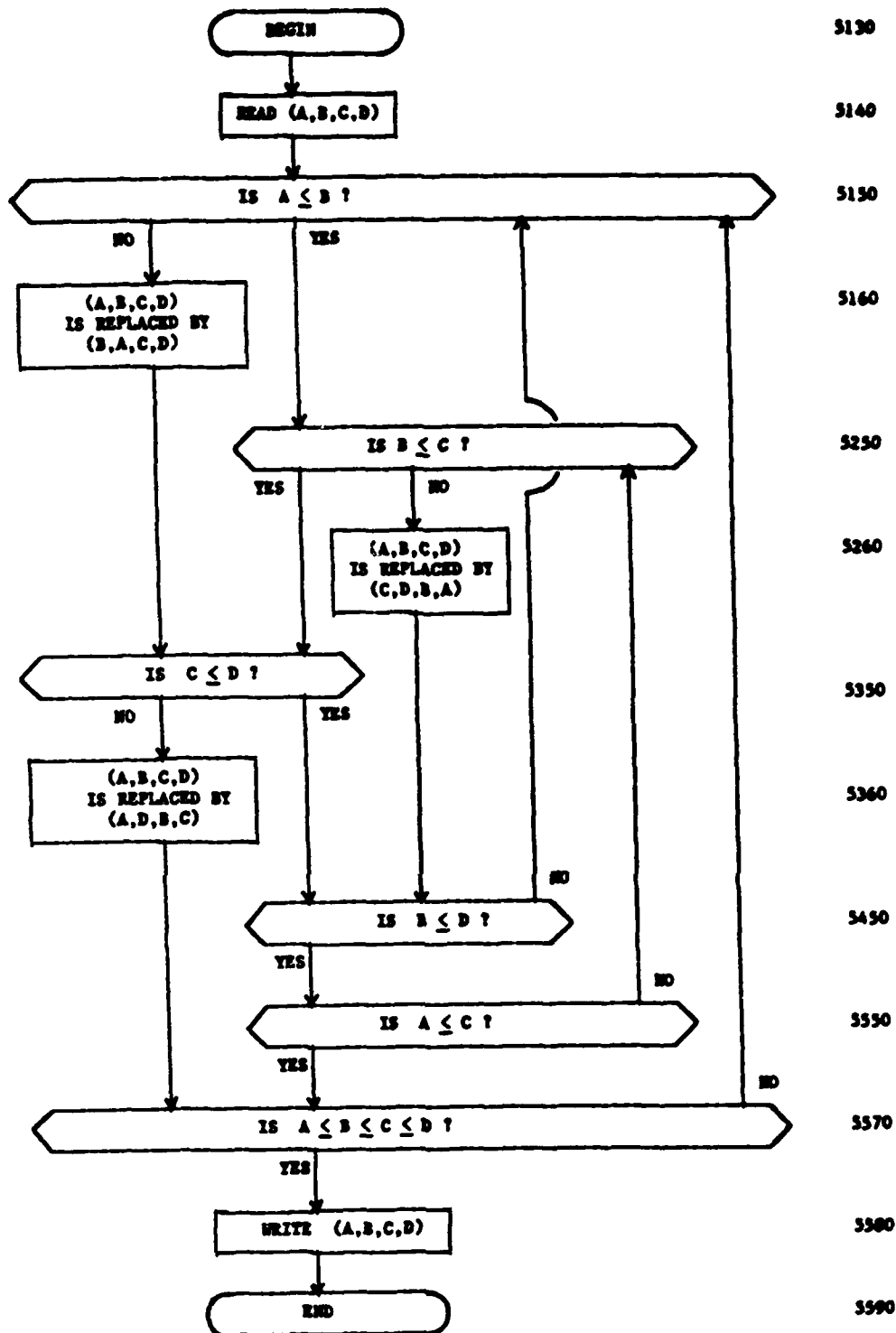
Figure 3.2

A flow diagram which defines the K(5) ZENSORT algorithm

statements can be made precise and proved. In this way we can verify program correctness by reducing an infinite problem (dealing with all members of the three dimensional real vector space $R^{(1,2,3)}$ of lists $(r(1),r(2),r(3))$ of three real numbers, or with all members of $R^{(1,2,3,4)})$ to a finite one (dealing with all members of $\{1,2,3\}^{(1,2,3)}$, or of $\{1,2,3,4\}^{(1,2,3,4)})$. This verification artifice is in keeping with the oft-noted [WI66; DI68; KE78] difficulty of understanding and analyzing the content of nonplanar flow diagrams. Our method of verification of correctness (ignoring the content and readability of the description of the algorithm and merely running it on a finite, but sufficiently representative, class of examples) contrasts with the method structured programming seems to suggest (looking at the content of the algorithm expressed by a planar flow diagram and using this understanding of the content and meaning of the algorithm to verify its correctness.) It is important to be aware of one other feature of these two routines. If the list (3,1,2) is input to K(3,3) ZENSORT the control flows along every edge of the flow diagram and through every vertex (i.e. box) before the answer (1,2,3) occurs as output. Similarly, if the list (3,2,1,4) is input to K(5) ZENSORT the control flows along every edge and through every vertex before the answer (1,2,3,4) occurs as output.

Why are the flow diagrams shown in Figures 3.1 and 3.2 intrinsically nonplanar? The answer is that they are derived from the two basic nonplanar Kuratowski graphs K(3,3) and K(5) in the following obvious sense. Take the graph [BE71, pp. 1-8] underlying [BE71, p. 220] the K(3,3) ZENSORT flow diagram, for example. Contract this graph by lumping boxes 3130, 3140 and 3150 together (collapsing the edges between) to form node 31, and by lumping boxes 3650, 3670 and 3680 together to form node 36. Then node 32 replaces box 3260, 33 replaces 3350, 34 replaces 3460 and 35 replaces 3550. At this point Figure 3.1 has been transformed into Figure 3.3. In the same fashion it is obvious how to turn Figure 3.2 into Figure 3.4.

At this point we can give the definition of a snarl. A snarl is a subdigraph [BO76, p. 171] of a flow diagram whose underlying graph can be contracted to either a K(3,3) or a K(5), and is minimal with respect to this property. Thus Figure 3.1 is not a snarl. But if you remove boxes 3130, 3140, 3670 and 3680 from it (together with the four obviously corresponding edges), the remaining boxes and edges form a snarl. Similarly Figure 3.2 without boxes 5130, 5140, 5580 and 5590 (and, of course without any of the four edges which touch any of these four boxes) is a snarl. Loosely, then, a snarl is a minimal nonplanar part of a flow diagram.

The ZENSORT flow diagrams can be characterized as GOTOful, in the sense that any program written from them must have an unconditional branch somewhere. In K(5) ZENSORT in Figure 3.2, for example, we have drawn a crossover in the control flow from the comparison box 5450 to the comparison box 5150. Somebody coding from the rendering of K(5) ZENSORT contained in Figure 3.2 might therefore be inclined to put the GOTO after the comparison IS B $\leq$ D ?. But there are many other ways to display the same flow diagram. And they would suggest other placements for the GOTO. The point is that the GOTO has to be in the code somewhere.

To summarize the digression, there is a gap in the literature, which the ZENSORT routines will now fill. Nonplanar flow diagrams exist. So it
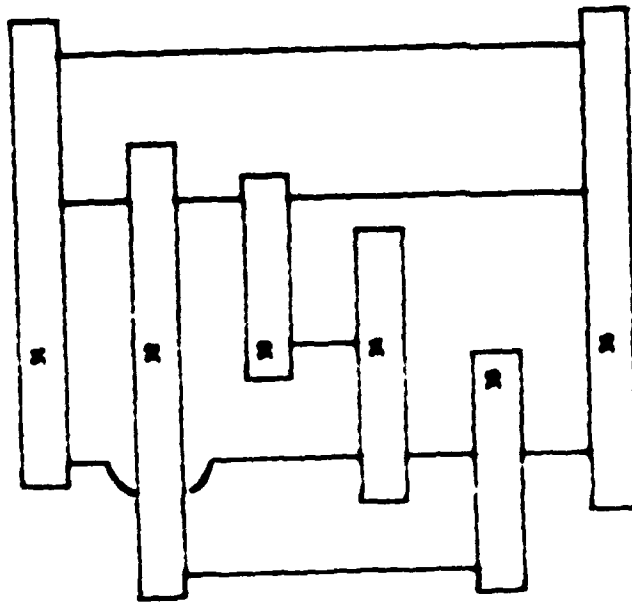
Figure 3.4.

A Kuratowski K(5) graph obtained by contracting the underlying graph of the flow diagram shown in Figure 3.2. This contraction identifies boxes 5130, 5140, 5150 and 5160, boxes 5230 and 5260, boxes 5350 and 5360, and boxes 5550, 5570, 5580 and 5590.

Figure 3.3.

A Kuratowski K(3,3) graph obtained by contracting the underlying graph of the flow diagram shown in Figure 3.1. This contraction identifies boxes 3130, 3140 and 3150. It also identifies boxes 3650, 3670 and 3680.
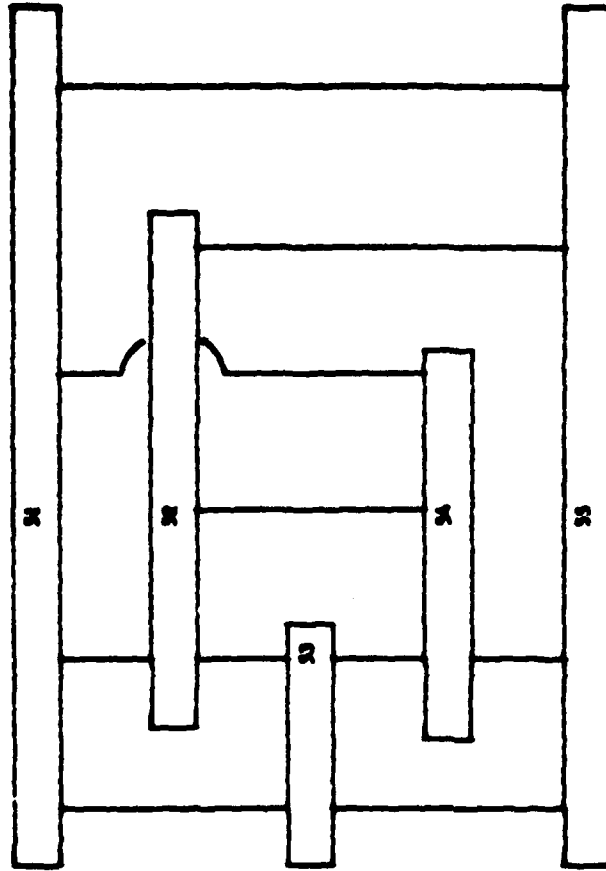
is false to say that all code with unconditional branching is merely bad code, and can be rewritten. Probably most such code up to now is, in fact, bad code in need of rewriting. But it is now clear that there are programs from which it is impossible to remove a GOTO. This is not to deny the existence of artifices to use a given program as a basis for producing a related GOTOless program. Many such [KN74] are known. But, in a sense which it would take too long to make precise within the confines of this proposal, those artifices produce merely a related algorithm, not an algorithm fully equivalent to the one specified by the original nonplanar flow diagram. In consequence of this fact this proposal departs from the popular posture of merely viewing GOTO with disdain. Instead it accepts snarls, and the corresponding GOTOful code, as interesting objects of investigation in their own right and as essential boundary conditions on any attempt to produce algorithms capable of producing useful flow diagrams or in other ways structuring software design on the basis of merely local information about control flows.

The digression is now over.

3b. The problem and the possibilities.

The problem we wish to address in this proposal, then, is this. At present there is no automated way to take pieces of local information (which relate a single instruction, question or subroutine to the objects it interfaces with) and integrate them into a global picture of the control flows in a program or larger piece of software. This proposal is for a project which will solve the problem. The problem is three dimensional, not planar, in the sense that not all flow diagrams for programs have underlying graphs which are imbeddable in the plane. Every graph is, of course [WI79, pp. 22-23], imbeddable in Euclidean 3-space. Being seemingly first to recognize this complication, YLYK Ltd. is able to avoid pursuit of the impossible dream of taking a planar approach to producing understandable visual representations of the global structure of pieces of software.

In the long term the following goals should be achievable:

Goal 1. Using appropriate local information, to produce a global "solid" flow diagram (i.e. a very special kind of representation of a digraph in 3-space);

Goal 2. Using appropriate local information, to determine whether there is a planar equivalent of this solid flow diagram;

Goal 3. To produce a drawing on paper of this planar flow diagram if it exists;

Goal 4. To produce an optimal drawing on paper of a flow diagram if that diagram is nonplanar. An optimal drawing is one which has a minimal number of crossovers;

Goal 5. To move from local information to pictorial representations (3-dimensional if necessary) of the flow diagram which are extremal in any one of a variety of ways. One type of extremality would be a display which would suggest a maximally parallel implementation which would buy speed at the cost of using several processors. Another type of extremality is shown by Figure 3.1 which is drawn so that all downward control flows are on the right half-page. In Figure 3.2 all downward flows are on the left. Such representation techniques may help in

understanding a program written from, or corresponding closely
to, such a flow diagram.

Goal 6. To examine nonplanar flow diagrams in their own right with a view
to understanding the kinds of programs which correspond to
them. During the last twenty years we have learned a lot about
how to read and write "good" structured (i.e. planar) programs.
Now that we know that nonplanar structures exist, and cannot be
made to go away, it is time to seek for a comparable improvement
in our ability to understand snarls and their relationship to the
larger structures they reside in;

Goal 7. To examine "batching", an analog of pipelining which is
appropriate to snarls (as we shall briefly indicate in Section 7)
and which promises manyfold speedup of programs involving snarls;

Goal 8. To ascertain whether some biological systems act in ways which
seem naturally to correspond to code with snarls, and attempt to
give examples of code with snarls which is in some way superior
to planar code for carrying out the same task;

Goal 9. To examine whether unbounded cellular automata, which are more
general [WO55] than Turing machines, can be used as a basis for
"3-dimensional" computer languages which need no GOTOs. In this
way we might be able to exorcise GOTO after all. But the process
would be general and scientific, rather than merely moralistic
("The writing of structured programs is a mark of good taste. So
write them!") or improperly based (on unjustified planarity
assumptions about flow diagrams).

These nine points are a tall order. We must now sort them out into
Phase I, Phase II and Blue Sky.

## 4. Phase I technical objectives

### 4a. An example of unfolding a flow diagram.

As in Section 3, we must digress to consider an example. The reason
is the same as in Section 3: the novelty of the material, and the
necessity to have a concrete example of what is desired and the sense in
which it is possible. We have seen examples of (truly) nonplanar flow
diagrams. Many diagrams one sees in the literature, however, are merely
inept renditions of planar flow diagrams. The renditions in question are
hastily drawn and consequently have crossovers. It is important to remedy
these flaws in existing planar flow diagrams and see to it that they do
not creep into our drawings of future flow diagrams if those diagrams are
planar in the graph theoretic sense. Recent YLYK Ltd. experiments with
simplifying flow diagrams — this proposal will speak of "unfolding"
diagrams hereinafter — have led to a viewpoint partly analogous to and
partly at variance with Dijkstra's attitude [DI76b; JE81] toward code, in
the sense of a program written in ADA, FORTRAN, some assembly language or
what have you. So this proposal will speak of "unfolding spaghetti
diagrams" to produce "structured diagrams", concepts which we will make
more explicit below. In one regard, this approach embodies higher hopes
for flow diagrams than Dijkstra has for code. He seems to feel that it is
easy to go from bad spaghetti code to badly suboptimal repulsive
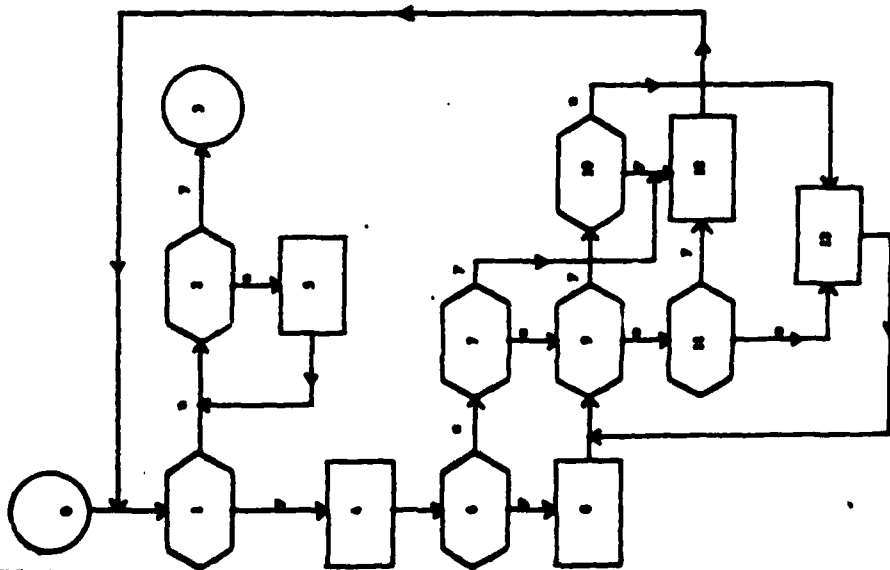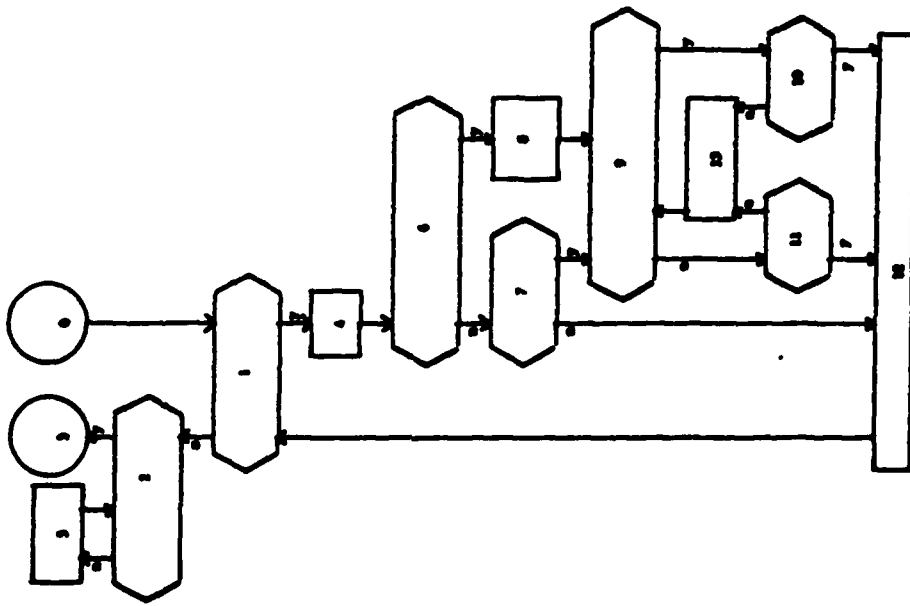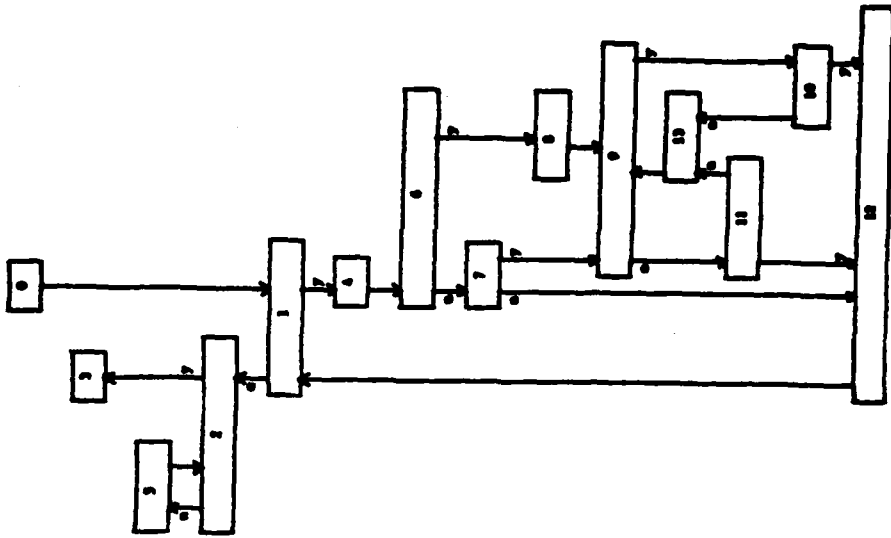
Figure 4.3

Figure 4.2

Figure 4.1

hard-to-understand structured code. It seems so far that the unfolding process, even starting from quite ugly spaghetti flow diagrams, should produce fairly satisfactory "structured" flow diagrams. The reason for this is that a flow diagram for process P has far fewer pieces (and most of the pieces are more naturally related to P) than a working program in any language in which programmers write actual running code today. In another regard, our expectations are "lower" than Dijkstra's. There will be no exorcism of "unstructured" flow diagrams (i.e. of flow diagrams which cannot be embedded in the plane). Such nonplanar objects, characterized by the presence of snarls, are here to stay. They must be understood in their own right, and may turn out to be important objects in computer science.

To set the stage let us begin with an example, an ostensibly nonplanar flowchart taken from [EL73, p. 279] (with a small modification of no importance to the purposes of this proposal. The boxes contain only numbers, rather than questions or instructions, because the topology is all that counts). What one finds in [EL73, p. 279] is, essentially, Figure 4.1. It has flows going every which way (i.e. north, south, east and west). It has two crossovers (despite which fact we will see that it is planar). It has flows merging into each other at four locations. It is spaghetti. Now look at its unfolded version in Figure 4.2. It is now clearly planar (whence we see that the crossovers in Figure 4.1 were unnecessary and misleading). The flows go only north and south. There are no crossovers. Flows go from node to node in separate streams without joining one another. Figure 4.2 is the same collection of boxes and the same collection of flows. But Figure 4.2 makes more sense than Figure 4.1. It also exposes subroutines in a way obvious to the most superficial viewer. Obviously, also, 0 is a source (i.e. a box with no arrows pointing in toward it). Similarly every one of the following sets of boxes is a sink (i.e. a structure with no arrows pointing outward)

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\};$$
$$\{2, 5, 3\};$$
$$\{3\}.$$

These facts are obvious from Figure 4.2 or Figure 4.3 but not from the equivalent, but ineptly drawn, Figure 4.1.

This ends the digression.

4b. The general idea of unfolding a flow diagram.

With these examples in mind we set out to formalize the idea of unfolding a flow diagram. We begin with the adjacency matrix [RO64; BE71, pp. 41-43], or (Riemann) sets of a flow diagram. We let

$$\zeta(i,j) = 0$$

if there is no flow from box i to box j, but

$$\zeta(i,j) = 1$$

if there is a flow from box i to box j. The 14 by 14 adjacency matrix of Figure 4.1 (or, what is the same thing, of Figure 4.2) is

|  |  | column indices | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| r o w   i n d i c e s | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|  | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
|  | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
|  | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
|  | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|  | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|  | 12 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Its zeroth column (i.e. its initial column) is the zero column (i.e. column with no nonzero entries) because box 0 is a source. Its row 3 (the fourth of its rows) is a zero row because box 3 is a sink. It has 20 nonzero entries because there are 20 flows. Its diagonal is zero because there are no loops in a flow diagram. The matrix $\zeta$ is a purely local description of the flow. The first objective of this research is to attain Goal 1 by producing an integrating operator $\int$ capable of producing something like a three dimensional version of Figure 4.2. Figure 4.3 is just a stretched-out version of Figure 4.2. The idea is that assigning each box a different height might make it easier to express the algorithm $\int$.

It would be desirable to have $\int$ act so that

$$\int \zeta = \text{Figure 4.3} \ .$$

More specifically, we want a simple algorithm for $\int$. Given a square incidence matrix $\zeta$ (You can, of course, always write down $\zeta$ from a flow diagram, no matter how spaghetti the flow diagram is, or even from any complete verbal description of what is to be done), we want an algorithm which produces a series of horizontal boxes, each box at a different height, and a series of vertical flows between pairs of boxes as in Figure 4.3.

We also want to attain Goal 3. If the flow diagram is planar (i.e. if its underlying graph is imbeddable in the plane) we want a variant of the $\int$ algorithm to draw a plane imbedding in the form of a pictorial representation without crossovers.

The staff of YLYK Ltd. has never failed to unfold any challenge flow
diagram taken from the technical literature (i.e. has not failed to find
an equivalent planar flow diagram without crossovers), which is why it was
led to construct K(3,3) ZENSORT and K(5) ZENSORT.

A conjectural explanation for this ostensible absence of nonplanar
flow diagrams prior to the two ZENSORT routines is that disciplined
scientific minds tend to think in "Western" and "linear" fashions which
are not sufficiently convoluted to produce nonplanar flow diagrams.
Perhaps most people think like Wirth [WI79], in other words, only not as
capably. They refine routines by breaking tasks down and wind up with
code which is GOTOless in spirit, even though stylistically defective
[KE78] and in need of reworking to make it GOTOless in fact.

Goal 2 may be harder to achieve algorithmically. YLYK Ltd. will
devote some time to it, but will divert its efforts to other goals, or
will accept only partial attainment of Goal 2, if extreme difficulties
arise. By partial achievement of Goal 2 we mean an algorithm which
sometimes establishes planarity, sometimes establishes nonplanarity and
sometimes halts without establishing either, but which always halts
within, let us say, an hour.

The fourth Phase I objective is attaining Goal 4. This appears to be
so like Goal 3 that we merely repeat what we have said about Goal 3. Goal
5 will also be met part way (via a routine for suggesting highly — though
perhaps not maximally — parallel implementation of any given algorithm).

At any rate the ability to move from $\zeta$ to a pictorial rendition of a
flow diagram which is planar, or which has a maximal number of parallel
flows (presumably sometimes useful for speedy parallel implementation), or
which has a minimal number of parallel flows (presumably sometimes easier
to decompose conceptually into subroutines and, thus, perhaps easier to
understand), would seem to be useful to people building algorithms as well
as people trying to understand existing algorithms. Such problems are
timely, hard [VI85; HU82] and can exhibit combinatorial explosion, but it
would be worthwhile to have even close-to-optimal solutions available, and
there are encouraging related [TA82; HO84] results. Consequently YLYK
Ltd. will devote considerable effort to Goal 5. But it will accept
limited success in the sense that an algorithm for producing a highly
parallel (rather than maximally parallel) implementation may be the only
result attained in Phase I.

Some progress toward the attainment of Goals 6, 7, 8, or 9 is likely
to occur in Phase I, but only incidentally. No formal work on them is
proposed for Phase I.


## 5. Phase I work plan.

The first objective in Phase I, Goal 1, is straightforward and
virtually certain of attainment within 6 months. This is the production
of an algorithm $\int$ which takes the adjacency matrix $\zeta$ of a flow diagram as
its input. The output, $\int \zeta$, of $\int$ will be two sets. The first set
(called the set of vertices) is a collection of horizontal cells (i.e.
parallelograms lying in planes parallel to the $xy$ plane and having sides
parallel to the $x$ axis and the $y$ axis) at pairwise unequal positive
integer heights (The single z-coordinate common to all points in a single

horizontal cell will be called the height of the cell). The second set
(called the set of edges) will be a set of ordered pairs of points of the
form (h,t) = ( (h[1], h[2], h[3]), (t[1], t[2], t[3]) ) where

$$h[1] = t[1]$$

$$h[2] = t[2]$$

and where the head point h of the edge (h,t) lies in one vertex and
the tail point t of the edge (h,t) lies in another vertex. Thus the
edges amount to directed vertical line segments connecting one vertex to
another. A vertex (h,t) must also have the property that the line
segment L((h,t)) joining h to t will not intersect any vertex other
than the one containing h and the one containing t.

The structure $\int \zeta$ so produced is the most useful standard form of
an imbedding of the flow diagram described by $\zeta$ in real three
dimensional Euclidean space. It consists of horizontal boxes and vertical
flows in a three dimensional analogy to Figure 4.3.

The simplicity and robustness of the available constructive proofs
[WI79, pp. 22-23] of the imbeddability of graphs in 3-space, coupled with
the power of the Wagner/Fary [BE71, pp. 84-85] approach to imbedding
simple graphs using straight lines, make success in the first objective
virtually a foregone conclusion. It remains to be seen, of course, how
fast the algorithm is. By all indications it will be a (deterministic)
polynomial time algorithm which can deal with 100 by 100 matrices $\zeta$ for
just a few dollars. One of the reasons for believing that costs will be
low (even if the complexity class of $\int$ is worse than P) is that it is
very hard to conceive of a humanly producible flow diagram whose digraph
adjacency matrix is nonsparse [GO83, p. 6] once the number of boxes (i.e.
vertices) rises toward the hundreds.

The second objective in Phase I, Goal 2, is more iffy. This
objective is to produce an algorithm $\Delta$ which takes the adjacency matrix
$\zeta$ of a flow diagram as an input. The output $\Delta\zeta$ of $\Delta$ will be 2 if
$\Delta$ can economically prove that $\zeta$ is the adjacency matrix of a planar
digraph, will be 3 if $\Delta$ can economically prove that $\zeta$ is the adjacency
matrix of a nonplanar digraph, and will be 4 if $\Delta$ determines that it
cannot output 2 or 3.

One such algorithm is PRINT "4", of course. The question is to what
extent it is possible to improve on it. There are many ways to recognize
planarity in a graph (the loopless digraph problem is a bit harder). One
of the most promising, from an algorithmic standpoint, is Whitney's
criterion [HA71, p. 115], namely the existence of a combinatorial dual.
The nonuniqueness of such a dual is a plus for a designer producing code
aimed at outputting a 2. Recognizing nonplanarity is different. It would
seem desirable to have a means of finding submatrices of $\zeta$ which look
like the adjacency matrices of snarls. So far the only advantage YLYK
Ltd. can bring to the search is that it knows snarls exist.

The third objective in Phase I is Goal 3. The idea behind it is as
follows. Suppose that the algorithm $\int$ exists. Suppose that a flow
diagram whose adjacency matrix is $\zeta$ is known to be planar, i.e. that
some algorithm $\Delta$ outputs $\Delta\zeta = 2$. (This is a weaker assumption than
that a pretty good algorithm $\Delta$ exists. Some oracle might just tell us
that this particular $\zeta$ is the $\zeta$ of a planar flow diagram). Produce an
algorithm $\Pi$ which takes the triple

$(\zeta, \int \zeta$, the fact that there is a $\Delta$ such that $\Delta\zeta = 2)$

as input, and produces an output

$\Pi(\zeta, \int\zeta$, the fact that there is a $\Delta$ such that $\Delta\zeta = 2)$

(which we will abbreviate as $\Pi\zeta$). This output should be a two dimensional version of $\int\zeta$. It will have vertices which are horizontal line segments (no two of them at the same height) in the xy plane and directed vertical edges. There will be no crossovers. It will, in short, look like Figure 4.3. This is an interesting problem, perhaps solvable in principle by somebody who has the resources to build a solid model of $\int\zeta$ in Euclidean 3-space and the leisure to examine it from every possible vantage point therein (this approach seems to entail the use of both projective and affine geometry) to ascertain whether there was such a vantage point from which it appeared planar. Such an approach might fall anywhere between simple linear algebra and sophisticated computer graphics, but would more likely be the former and would very likely lead to an algorithm $\Pi$ with extremely low computational complexity or else to an algorithm without provably low complexity but with high probability of quickly producing an output planar flow diagram.

As noted in Section 5, it is anticipated that Goal 1 (a working routine for forming $\int\zeta$ ) and Goal 3 (a working routine for drawing $\int\zeta$ in a plane with no crossovers if $\zeta$ is the adjacency matrix of a digraph whose underlying graph is planar) will have been attained by the end of Phase I.

The fourth Phase I objective is Goal 4. This should be the easiest goal to meet.

As noted in Section 4, the fifth Phase I objective is Goal 5 but YLYK Ltd. would be content with limited success in regard to one extremal property, parallelizability. This objective will be sought only after a working routine for turning $\zeta$ into $\int\zeta$ exists. Here no powerful mathematical tools are known, but several people who will be employed on the project have begun to acquire working familiarity with what is involved by working out examples.

## 6. Related work. Bibliographic citations list.

### 6a. Scientists who will work on the project.

Bob Blakley, the principal investigator on this SBIR Phase I proposal, served as a draftsman for the City of Bryan, Texas, in the summer of 1978. He is an expert scientific programmer, having been employed at various times over the last eight years in software production and maintenance by research contracts and grants in the Mathematics, Mechanical Engineering, Statistics, Chemistry, Biochemistry and Biophysics departments of Texas A&M University, the Geophysical Fluid Dynamics Laboratory at Princeton University, the University of Michigan Computer Center, the Winterhalter Corporation of Ann Arbor, Michigan, as well as for YLYK Ltd. of Ann Arbor, Michigan. He has had extensive experience in algebraic scientific software production, some of it in collaboration with G. R. Blakley. He has a substantial academic background in mathematics, logic, computer science and natural languages. He is conversant with a dozen computer languages, several of which are assembly languages, and with IBM PC software/hardware interface(BIOS). He was Principal Investigator on YLYK Ltd. Phase I SBIR Contract F 49620-83-C-0160 with

AFOSR (duration 6 months, beginning 30 September 1983). The research on the AFOSR contract in question dealt with high-speed low-cost ways to get a message from a sender to a receiver when some channels linking them become inoperative.

G. R. Blakley, R. D. Dixon and A. M. Hobbs will be employed as consultants on this work.

Dr. G. R. Blakley invented threshold schemes, and is a major contributor to their theory. His interest in linear algebra, combinatorics, and their applications outside mathematics, goes back twenty years, and has issued in numerous publications. He has had 30 years acquaintanceship with computers and has recently completed three years as Principal Investigator on a National Security Agency grant to do unclassified research in information theory.

Dr. R. D. Dixon is a computer scientist with doctoral training and numerous publications in mathematics and computer science. His interest in linear algebra, combinatorics, and their uses in computer science goes back twenty years.

Dr. A. M. Hobbs is a graph theorist with extensive experience in computing. He has some 20 publications.

Bob Blakley, G. R. Blakley, R. D. Dixon and A. M. Hobbs have all known each other for many years. They communicate effortlessly with each other on technical matters. See Sections 9 and 11 below for more on these individuals.

6b. Bibliographic citations list.

AD82  W. R. Adrion, M. A. Branstad and J. C. Cherniavsky, Validation, verification, and testing of computer software, ACM Computing Surveys, Vol. 14 (1982), pp. 159-192.

BA72  F. T. Baker, Chief programmer team management of production programming, IBM Systems Journal, Vol. 11, No. 1, Jan. (1972), pp. 56-73.

BE71  M. Behzad and G. Chartrand, Introduction to the Theory of Graphs, Allyn and Bacon, Boston (1971).

BE74  H. Behnke, F. Bachmann, K. Fladt and W. Süss, Fundamentals of Mathematics, Volume 1, MIT Press (1974).

BE85  E. A. Bender and J. T. Butler, Enumeration of stuctured flowcharts, Journal of the ACM, Vol. 32 (1985), pp. 537-548.

BL8  G. R. Blakley, IEEE Transactions on Computers, Vol.   (198 ), pp.

BO66  C. Böhm and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, Communications of the ACM, Vol. 9 (1966), pp. 366-370.

BO76  J. A. Bondy and U. S. R. Murty, Graph Theory with Applications, American Elsevier, New York, and Macmillan, London (1976).

BU74  D. Butterworth, Letter to the Editor, Datamation, Vol. 20, No. 3, (1974), p. 158.

CO86  D. I. A. Cohen, Introduction to Computer Theory, John Wiley and Sons, New York (1986).

DA72  O. -J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press (1972).

DI65      E. W. Dijkstra, Programming considered as a human activity, Proceedings of the IFIP Congress (1965), pp. 213-217.

DI68      E. W. Dijkstra, GO TO statement considered harmful, Communications of the ACM, Vol. 11 (1968), pp. 147-148.

DI76      E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey (1976).

DO73      J. R. Donaldson, Structured programming, Datamation, Vol. 19, No. 12 (1973), pp. 52-54.

EL73      M. Elson, Concepts of Programming Languages, Science Research Associates, Chicago (1973).

GO83      G. H. Golub and C. F. Van Loan, Matrix Computations, Johns Hopkins University Press, Baltimore (1983).

HA69      F. Harary, Graph Theory, Addison-Wesley, Reading, Massachusetts (1967).

HO82      W. E. Howden, Validation of scientific programs, ACM Computing Surveys, Vol. 14 (1982), pp. 193-227.

HO84      H. J. Hoover, M. M. Klawe and N. J. Pippenger, Bounding fan-out in logical networks, Journal of the ACM, Vol. 31 (1984), pp. 13-18.

HU82      H. B. Hunt, III, On the complexity of flowchart and loop program schemes and programming languages, Journal of the ACM, Vol. 29 (1982), pp. 228-249.

JA75      M. A. Jackson, Principles of Program Design, Academic Press, London (1975).

JE79      R. W. Jensen and C. C. Tonies, Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey (1979).

JE81      R. W. Jensen, Structured programming, IEEE Computer Magazine, Vol. 14, No. 3, March (1981), pp. 31-48.

KA74      R. A. Karp, Letter to the Editor, Datamation, Vol. 20, No. 3, (1974), p. 158.

KE78      B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, Second Edition, McGraw-Hill, New York (1978).

KN74      D. E. Knuth, Structured programming with GOTO statement, ACM Computing Surveys, Vol. 6 (1974), pp. 261-301.

LA66      S. Lang, Linear Algebra, Addison-Wesley, Reading, Massachusetts (1966).

MA67      S. MacLane and G. Birkhoff, Algebra, MacMillan, New York (1967).

MA77      Y. I. Manin, A Course in Mathematical Logic, Springer-Verlag, New York (1977).

MI72      H. D. Mills, Mathematical foundations for structured programming, IBM Technical Report FSC 72-6012, International Business Machines Corp., Gaithersburg, Maryland, February (1972).

MO76      J. D. Monk, Mathematical Logic, Springer-Verlag, New York (1976).

PE73      W. W. Peterson, T. Kasami, and N. Tokura, On the capabilities of while, repeat, and exit statements, Communications of the ACM, V 1. 16 (1973), pp. 503-512.

QU84      M. J. Quinn and N. Deo, Parallel graph algorithms, ACM Computing Surveys, Vol. 16 (1984), pp. 319-348.

RO64      G. -C. Rota, On the foundations of combinatorial theory, I. The Theory of Möbius functions, Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete, Vol. 2 (1964), pp. 340-368.

TA82      K. Takamizawa, T. Nishizeki, and N. Saito, Linear-time computability of combinatorial problems on series-parallel graphs, Journal of the ACM, Vol. 29 (1982), pp. 623-641.

VI85      U. Vishkin and A. Wigderson, Trade-offs between depth and width in parallel computation, SIAM Journal on Computing, Vol. 14 (1985), pp. 303-312.

WI66      N. Wirth and C. A. R. Hoare, A contribution to the development of ALGOL, Communications of the ACM, Vol. 9 (1966), pp. 413-432.

WI71      N. Wirth, Program development by stepwise refinement, Communications of the ACM, Vol. 14 (1971), pp. 221-227.

WI74      N. Wirth, On the composition of well-structured programs, ACM Computing Surveys, Vol. 6 (1974), pp. 247-259.

WI79      R. J. Wilson, Introduction to Graph Theory, Second Edition, Academic Press, New York (1979).

WO85      S. Wolfram, response to a question asked at Crypto '85, Santa Barbara, California, August (1985).

7. Relationship with future research, or research and development.

    7a. Anticipated results of Phase I effort if the project is successful.

    Suppose, first, that the Phase I work succeeds in attaining Goals 1,2,3 and 4 in a completely general and satisfactory way. Consider what will then be available to anybody with a well-determined program specification which has already progressed to the stage where, for each individual process, it is known which processes it immediately depends on, and which processes immediately depend on it (i.e. to anybody with complete local information $\zeta$ ). Such an individual will have available a cheap, fast computer program which will produce a complete global picture $\int \zeta$ of the structure of the program. In the planar case this means there will be a flat drawing $\int \zeta$ with horizontal boxes (within each box will be written either an operation or an interrogation) and vertical arrows (describing control flows from box to box), and this drawing will have no crossovers. See Figure 4.3 for an example.

    In the nonplanar case the cheap, fast program will produce a proof of nonplanarity. It will also produce two other structures, a solid one and a flat one. The solid structure will resemble the skeleton of a skyscraper under construction. It will consist of horizontal "floors" (i.e. horizontal rectangles which play the same role the boxes did in the flat drawing above, meaning that each "floor" has an operation or an interrogation written on it) and vertical "girders" (i.e. vertical arrows) describing control flows between "floors". A "girder" connecting "floor A" to "floor B" will never touch any other "floor".

The flat structure in this nonplanar case will be an arrangement of horizontal boxes and vertical flows, just as in the planar case above, with the following difference. There will be at least one place where a crossover occurs, i.e. where a vertical flow must tunnel under a horizontal box. See Figures 3.1 and 3.2 for examples. The number of such crossovers in the flat structure will be minimal.

Such representations are evidently much different from, and manifestly superior to, the sort of spaghetti that one sees in virtually any flow diagram of a complicated program in the literature to date.

It is in the nature of things that none of these pictorial representations of flow diagrams, whether planar or nonplanar, whether solid or flat, can be unique. This is a plus from two viewpoints. First, it will make it easier to build a fast algorithm for producing such a pictorial representation. Second, it will make these pictorial representations (all of which go by the name $\int \zeta$. Even though this is an abuse of language, it is in the spirit of calculus since indefinite integration does not produce a unique result) manipulable. Here is where the open-ended Goal 5 comes in. People will want to take some $\int \zeta$ and tweak it to produce more information about the algorithms underlying $\zeta$ or about how to implement those algorithms. It would clearly be desirable to modify an $\int \zeta$ drawing (or skyscraper skeleton) so as to be able to set up an extremally parallel (hence fast) implementation. And YLYK Ltd. will attempt to provide a routine for this purpose. But there will obviously be about as many desired types of tweaks as there are tweakers. So there will be literally no end to the sort of thing one can do toward attaining Goal 5.

7b. Significance of Phase I effort in providing a foundation for Phase II.

Phase II should be devoted to completing work on Goals 3 and 4, in the unlikely event anything remains to be done, as well as to completing work on Goal 2. It should also attain more subgoals associated with the open-ended collection of desired routines which make up Goal 5.

Goals 6, 7 and 9 are its proper areas of investigation. Goal 6, learning about routines containing snarls, is very extensive and very exciting. In a sense Goal 7, working out efficient batching, is merely a part of Goal 6. But batching is so promising it deserves a separate mention.

In pipelining it is often possible to push data through some crucial node at a rate of one item per clock-tick forever so that various items are at various stages of completion. This seems unlikely in snarls. Look at how data moves through $K(3,3)$ ZENSORT in Figure 3.1, for example. An input such as (3,1,2) goes through box 3150, and is transformed to (2,3,1), goes again through box 3150, is transformed to (2,1,3), goes again through box 3150, is transformed to (3,2,1), goes again through box 3150, is transformed to (1,2,3), and exits. This means that it is impossible to put an unending input data stream into box 3150, because at certain later times partially processed data must reenter it.

But it takes a while before a given item reenters box 3150. So it might be possible to put a few more operations (including NOP operations) into $K(3,3)$ ZENSORT in such a way as to force a sort of "uniform cycling time" on the algorithm. In this way one could then feed a batch of T inputs in, watch this batch move through, then feed another batch through, and so on. This is what is meant by the word "batching". In this way an

actual slowdown of parts of K(3,3) ZERSORT would lead to a 7fold speedup
of the algorithm as a whole.

It looks as though hatching is, rather naturally, more appropriate to
snarls than pipelining in which arbitrarily long data streams are fed in.

Goal 9, a "3-dimensional language" based on unbounded cellular
automata is speculative. If it is attained, it is not likely to be done
with just 4 or 5 man-years of effort. But it would be a true realization
of a worthwhile goal of structured programming, the complete and perfectly
general removal of the need for GOTO.

Goal 8 can be viewed as an open-ended blue sky sort of investigation
of the relationship between biology and the theory of algorithms. It will
probably not be attacked even in Phase II. The reason is that its high
intrinsic interest is not matched by obvious immediate utility.

8. Potential post applications.

8a. Potential commercial application of proposed project.

As noted in various places the research could, in the best possible
outcome of Phases I and II:

1. produce a new, completely general GOTOless "3-dimensional language";

2. massively modify and improve (perhaps revolutionize) our approach to
   software design;

3. provide a new and very different class of routines (those with
   snarls) for our study and use;

4. produce a major enhancement in our ability to make use of parallel
   computation.

8b. Potential utility to the Federal Government of proposed project.

The advantages cited in Subsection 8a are all of obvious use to the
Federal government. Since we are proposing general-purpose research it is
hard to see which of its results will be of more value to industry and
which of more value to government.

9. Key personnel.

YLYK Ltd. was incorporated in Delaware on 4 June 1979. It is
currently headquartered in Ann Arbor, Michigan. Its Employee
Identification Number is 22 2280588.

Bob Blakley, born 13 July 1960 in Washington D.C., is a citizen of
the U.S.A. and a 1982 honors graduate of Princeton University. He married
Karen Hejtmancik of College Station, Texas, on 7 August 1982. In 1984 he
received an M.S. degree in computer and communications sciences from the
University of Michigan. He is currently teaching a course on assembly
language and completing his doctoral dissertation in computer and
communications sciences at the University of Michigan. He is coauthor of
three papers on cryptography and information theory in Cryptologia, Volume
2 (1978), pp. 305-321, Volume 3 (1979), pp. 29-42, and Volume 3 (1979),
pp. 105-118. For 6 months, beginning on 30 September 1983, he was
principal investigator on an YLYK Ltd. SBIR Phase I contract with AFOSR
(High-speed low-cost ways to get messages from a sender to a receiver when
some channels linking them become inoperative, Contract No.
F 49620-83-C-0160). He is president of YLYK Ltd., and will be principal

investigator on the proposed research. His Social Security Number is
440-46-2353. See Section 6 above for more information concerning him.

10. Facilities/equipment/services.

The 3-room facilities available to TLTK Ltd. at Ann Arbor are
adequate to the task at hand. They provide the principal investigator
with a work area and necessary library and drafting facilities. Other
personnel can be accommodated there, or else assigned duties to be
performed on their own premises in consultant fashion. It is easy to
purchase computer time as needed in Ann Arbor and Detroit.

11. Consultants.

G. R. Blakley (Ph.D., Mathematics, University of Maryland, 1960) did
postdoctoral work at Cornell and Harvard. He has been on the mathematics
department faculty of the University of Illinois (Urbana), SUNY at
Buffalo, and Texas A&M University (where he was department head for many
years, and where he is currently a professor). He is author or coauthor
of more than 30 papers in mathematics and its applications, many of them
in linear algebra, complex variables and mathematical areas of computer
science. He is coeditor (with David Chaum) of Advances in Cryptology,
Proceedings of CRYPTO '84, Volume 196, Lecture Notes in Computer Science,
Springer-Verlag, Berlin (1985).

R. D. Dixon (Ph.D., Mathematics, Ohio State University, 1962) served
on the mathematics department faculty at the University of Illinois before
leaving to organize and head the mathematics department at Wright State
University in 1964. He subsequently helped to organize, and became head
of, the computer science department at Wright State, where he is now
professor. He has dozens of publications in linear algebra and analysis,
combinatorics and number theory, and in computer science.

A. M. Hobbs (Ph.D., Combinatorics, Waterloo, 1971) is a graph
theorist and a student of W. T. Tutte. He is the author of more than a
dozen papers in graph theory. His professional career has been largely at
the National Bureau of Standards and in the mathematics department of
Texas A&M University, where he is associate professor.

See Section 6 above for more on these consultants.

12. Prior, current or pending support.

No SBIR proposal, or other sort of proposal to the Federal Government
similar to this proposal, has been submitted or is under consideration.
If a proposal similar to this one is subsequently produced and submitted
to any agency of the Federal Government, TLTK Ltd. will promptly inform
the AFOSR Program Management Office.

**13. Cost proposal.**

### COST BREAKDOWN

### DEFENSE SMALL BUSINESS INNOVATION RESEARCH PROGRAM (SBIR) PHASE I

| | | |
|---|---|---|
| C1. | Offeror | YLYK Ltd. |
| C2. | Offeror's home office address | 2440 Stone<br>Ann Arbor, Michigan 48105 |
| C3. | Location where work<br>will be performed | Ann Arbor, Michigan |
| C4. | Title of proposed effort | Reduction of flow diagrams to<br>unfolded form modulo snarls |
| C5. | Topic number and topic title<br>from DOD solicitation brochure | AF 86-12<br>Research in Mathematics |
| C6. | Total dollar amount of the proposal | $ 49,973 |
| C7. | Direct material costs | $     0 |
| C8. | Material overhead | $     0 |

**C9. Direct labor**

9a. Principal investigator:
Bob Blakley, President, YLYK Ltd.
(rate: $24 per hour) 1040 hours                    $ 24,960

9b. Ancillary workers:
Technical typing/secretarial
Drafting/programming
(average rate: $9 per hour)
520 hours                                                        $   4,680
_____

9c. Estimated total direct labor                         $ 29,640

**C10. Labor overhead: 24% of direct labor**

Taxes (including FICA, FUTA, MESC),
Insurance, Employee benefits,
Accounting, Communications,
Facilities, Utilities                                          $  7,114

C11. Special testing                                        $     0

C12. Special equipment                                    $     0

C13. Travel:

    13a. Air transportation;
        5 round trips between points in
           the U. S. at $400 per trip      $ 2,000

    13b. Automobile rental;
        10 days at $40 per day        $   400

    13c. Per diem (lodging and meals);
        10 days at $70 per day        $   700

    13d. Conference and seminar
        registration fees            $   500

    13e. Estimated total travel        $ 3,600

C14. Consultants:

    14a. G. R. Blakley,
        (rate: $27 per hour) 96  hours   $ 2,592

    14b. R. D. Dixon,
        (rate: $27 per hour) 72 hours   $ 1,944

    14c. A. M. Hobbs,
        (rate: $19 per hour) 32 hours   $   608

    14d. Estimated total consultants   $ 5,144

C15. Other direct costs:

    Cost of renting computer time to carry
    out necessary computations       $ 1,000

C16. General and administrative expense:

    4 % of total direct costs
    incurred in items C9, C13,
    C14 and C15                $ 1,575

C17. Royalties                    $    0

C18. Fee or profit               $ 1,900

C19. Estimate of cost, and fee or profit:

    19a.  Direct labor           $ 29,640

    19b.  Labor overhead        $ 7,114

    19c.  Travel               $ 3,600

    19d.  Consultants          $ 5,144

    19e.  Other direct costs      $ 1,000

    19f.  General and administrative expense  $ 1,575

    19g.  Fee or profit          $ 1,900

    19h.  Total               $ 49,973

C20. Signature: _____

Bob Blakley, President, YLYK Ltd.

Date: 21 January 1986

C21. Offeror's answers to three questions.

21a. Has any executive agency of the United States government performed any review of YLYK Ltd.'s accounts or records in connection with any other government prime contract or subcontract within the past twelve months? No.

21b. Will YLYK Ltd. require the use of any government property in the performance of this proposal? No.

21c. Does YLYK Ltd. require government contract financing to perform this proposal contract? Yes. Advanced Payments. It is proposed that $15,000 be paid on Day 1. This will take care of startup costs, early consultant fees and travel, and certain direct labor expenses. It is, further, proposed that $10,000 be paid on Day 41, that $10,000 be paid on Day 81, that $10,000 be paid on Day 121, and that $4,973 be paid on receipt of the final rep

C22.     Type of contract proposed: Firm-fixed price.

[AG86]  Agha, G., Actors: A Model of Concurrent  Computation
and Distributed Systems, MIT Press, 1986.

[AH74]  Aho, A.V., Hopcroft,  J.E., and Ullman,  J.D.,  The
Design  and Analysis of Computer Algorithms, Reading, Mass.:
Addison-Wesley, 1974.

[BB85]  Bender,  E.A.,  and J.T.  Butler,  "Enumeration  of
Structured Flowcharts", JACM, 32:3(Jul. 1985), 537-48.

[BC71]  Behzad,  M., and  G. Chartrand, Introduction to  the
Theory of Graphs, Boston: Allyn and Bacon, 1971.

[BJ66]  Bohm, C.D., and G. Jacopini, "Flow Diagrams,  Turing
Machines,  and  Languages With Only Two  Formation  Rules",
CACM, 9:5(May 1966), 366-71.

[BO79]  Bollobas, B., Graph Theory: An Introductory  Course,
New York: Springer, 1979.

[BM76]  Bondy,  J.A., and  U.S.R. Murty, Graph  Theory  with
Applications, New York: Elsevier (North-Holland), 1976.

[BL76]  Booth,  K.S.,  and  G.S.  Lueker, "Testing  for  the
Consecutive  Ones  Property,  Interval  Graphs,  and  Graph
Planarity Using  PQ-tree  Algorithms", J. Comp. Sys.  Sci.,
13(1976), 335-79.

[CN85]  Chiba,  N., T. Nishizeki,  S. Abe, and T. Ozawa,  "A
Linear  Algorithm  for  Embedding  Planar  Graphs  Using
PQ-Trees", J. Comp. and Sys. Sci., 30(1985), 54-76.

[DE76]  Deo,  N.,  "Note  on Hopcroft and  Tarjan  Planarity
Algorithm", JACM, 23(1976), 74-5.

[DH83]  Duchet,  P., Y.  Hamidoune, M. Las Vergnas,  and  H.
Meyniel,  "Representing  A  Planar Graph by  Vertical  Lines
Joining Different  Levels", Discrete Mathematics  46(1983),
319-21.

[DI68]  Dijkstra, E.D., "Go To Statement Considered Harmful"
(letter), CACM, 11:3(Mar. 1968), 147-8.

[EV79]  Even, S., Graph Algorithms, Rockville, MD:  Computer
Science Press, 1979.

[ET76] Even, S., and Tarjan, R.E., "Computing an st-numbering", Th. Comp. Sci., 2(1976), 339-44.

[FA48] Fary, I., "On Straight-Line Representation of Planar Graphs", Acta. Sci. Math. Szeged, 11(1948), 229-33.

[FR81] de Fraysseix, H., and P. Rosenstiehl, Seminaire du Lundi, Paris, Dec. 1981. (cited in [DH83]).

[GJ79] Garey, M.R., and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, San Francisco: W.H. Freeman, 1979.

[GI85] Gibbons, A., Algorithmic Graph Theory, London: Cambridge, 1985.

[GR83] Goldberg, A., and D. Robson, Smalltalk-80: The Language and its Implementation, Reading, Mass.: Addison-Wesley, 1983.

[HO85] Hoare, C.A.R., Communicating Sequential Processes, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[HT74] Hopcroft, J.E., and R.E. Tarjan, "Efficient Planarity Testing", JACM, 21(1974), 549-68.

[ME84] Mehlhorn, K., Graph Algorithms and NP-Completeness, New York: Springer, 1984.

[MU75] Munkres, J.R., Topology: a first course, Englewood Cliffs, N.J.: Prentice-Hall, 1975.

[LE67] Lempel, A., S. Even., and I. Cederbaum, "An Algorithm for Planarity Testing of Graphs", Theory of Graphs, International Symposium, Rome, July 1966, P. Rosenstiehl, ed., Gordon and Breach, N.Y., 1967, 215-32.

[RE85] Reisig, W., Petri Nets: An Introduction, Berlin: Springer, 1985.

[RN77] Rheingold, E.M., Nievergelt, J., and Deo, N., Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[RU87] Rubin, F., "'GOTO Considered Harmful' Considered Harmful", CACM 30:3(Mar. 1987), 195-6.

[SA85] Sharp, J.A., Data Flow Computing, Chichester, West

Sussex, England: Ellis Horwood, 1985.

[SH8?] Sherlekar, D., Thesis, U. of Maryland, in preparation.

[TA71] Tarjan, R., "An Efficient Planarity Algorithm", Thesis, Stanford U., 1971.

[TA87] Tarjan, R., "Algorithm Design", CACM 30:3(Mar. 1987), 205-12.

[UL84] Ullman, J.D., Computational Aspects of VLSI, Computer Science Press, Rockville, MD, 1984.

[WS85] Wadge, W.W., and E.A. Ashcroft, Lucid: The Dataflow Programming Language, London: Academic Press, 1985.

[WA36] Wagner, K., "Bemerkungen zum Vierfarbenproblem", Jber. Deutsch. Math. Verein., 46(1936), 21-2.

[WI79] Wilson, R.J., Introduction to Graph Theory, 2nd Ed., New York, Academic Press, 1979.

END
7-87
DTIC